



Clipcode™ Source Tour For Angular 12

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

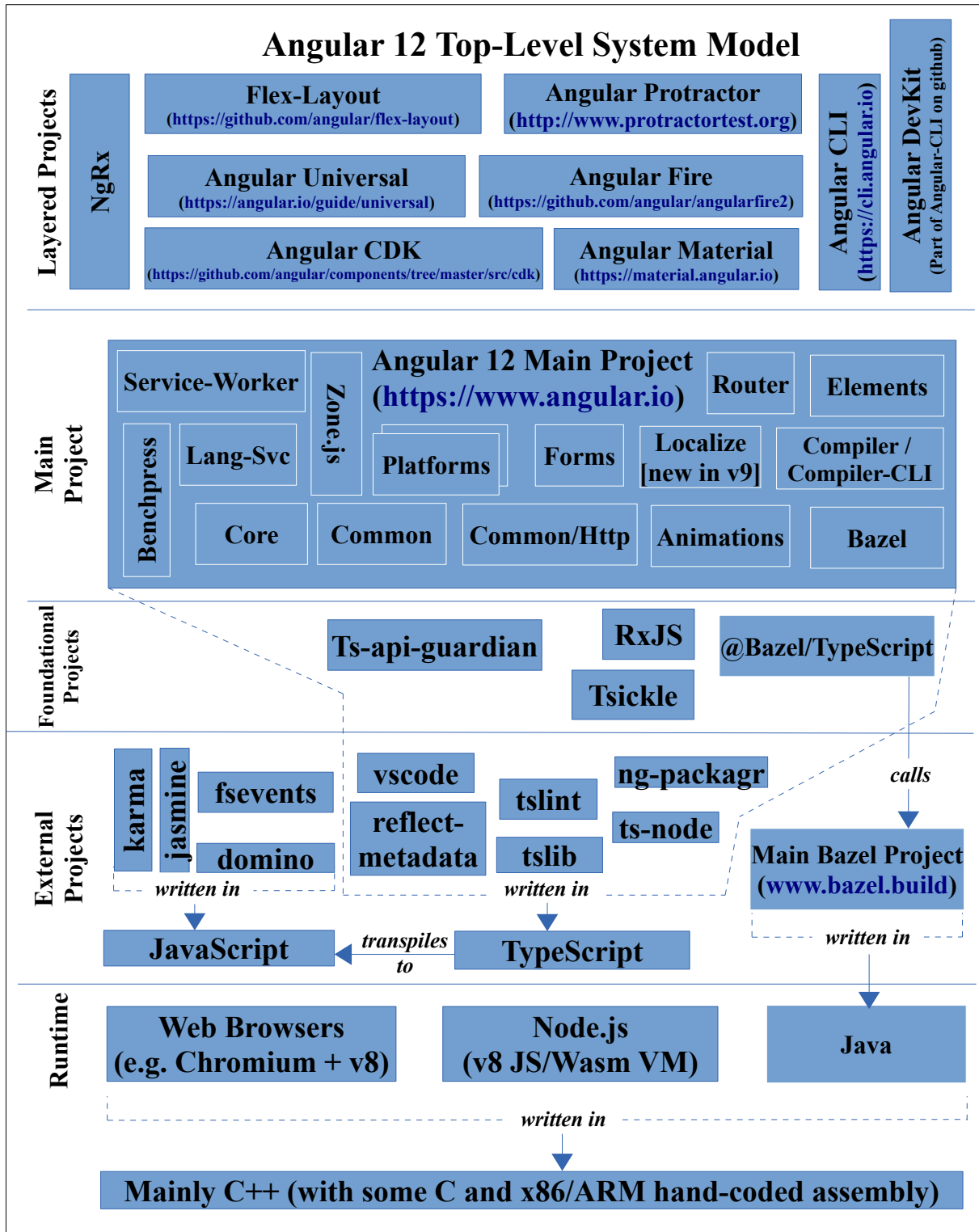
Copyright (c) 2021 Clipcode Limited - All rights reserved

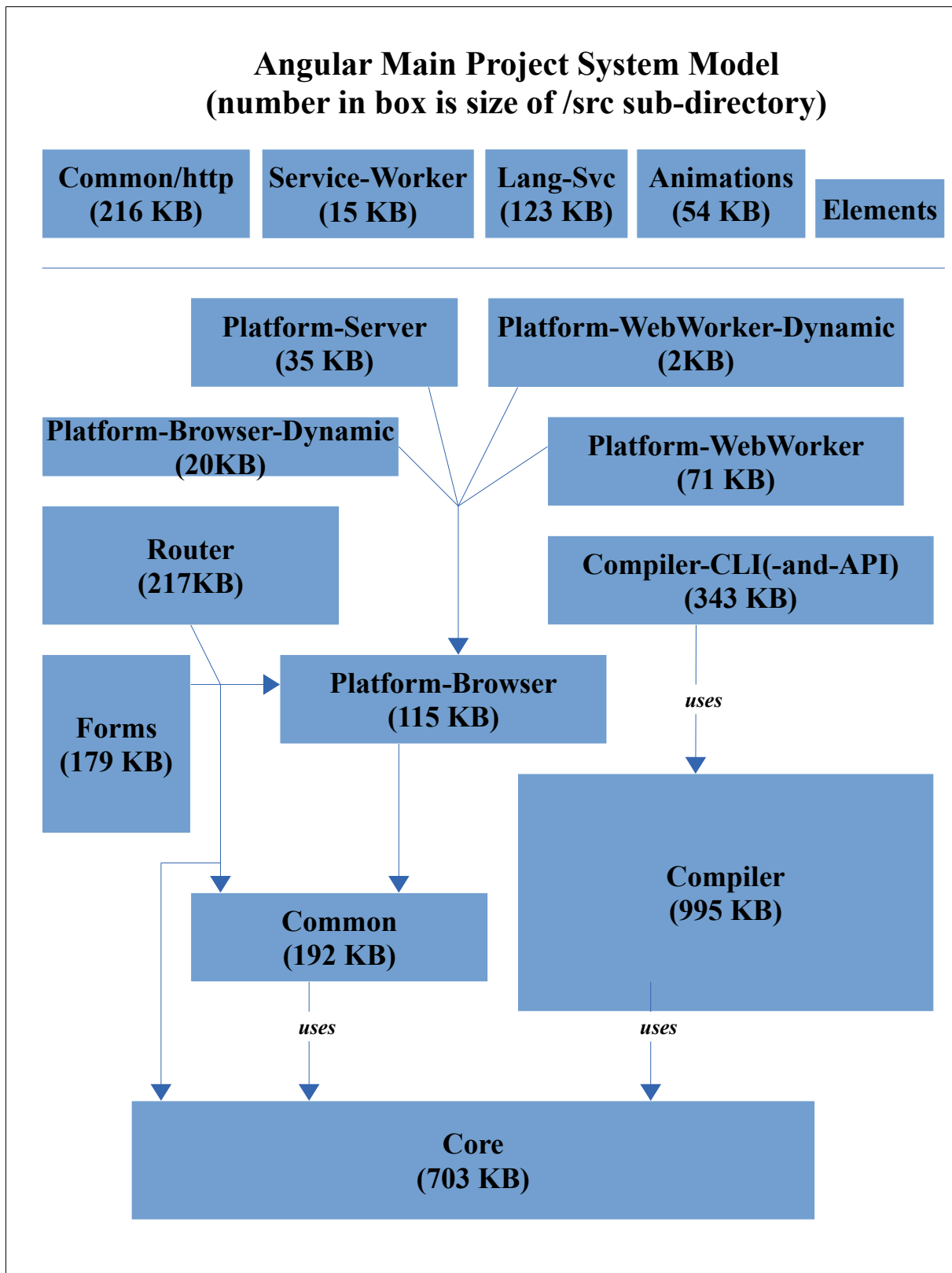
Clipcode is a trademark of Clipcode Limited - All rights reserved

Table of Contents

Preface.....	iii
1: Internals Of Zone.js.....	8
2: Tsickle.....	27
3: TS-API-Guardian.....	32
4: RxJS 6.....	37
5: The Core Package.....	40
6: The Common Package.....	86
7: The Common/HTTP Sub-package.....	98
8: The Platform-Browser Package.....	105
9: The Platform-Browser-Dynamic Package.....	128
10: The Platform-Server Package.....	133
11: The Platform-WebWorker Package.....	141
12: The Platform-WebWorker-Dynamic Package.....	168
13: The Router Package.....	170
14: Render3 (Ivy) in Angular.....	184
15: The Forms Package.....	221

Preface





Note JAVA is used by Bazel build system to build Angular itself – it is not needed at all when building your own Angular applications (unless you use Bazel, a good option for very large code bases). Note moment.js is a very popular date and time framework. We see it used by the [@angular/material-moment-adapter](https://github.com/moment/moment) package.

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. With a few exceptions, all of these projects are written with TypeScript (even the TypeScript compiler is itself written in TypeScript). The exceptions are the Bazel main project (which uses Java), fsevents for native access to macOS FSEvents along with the Jasmine unit test framework and Karma test runner (all these use JavaScript). The dependency on other projects varies.

Note that the Java-based Bazel main project is used to build Angular itself. Bazel and Java are not needed to build regular Angular applications. Bazel is great for incremental and multi-core builds so it has advantages when used with large codebases (such as Angular itself). In future Bazel is likely to become a sensible build option for larger Angular applications.

None of these 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.

- <https://code.visualstudio.com>

Of course you can use any 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 than 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>

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> (which is mostly quite good) but it is best studied in conjunction with the source code.

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, how it is optional (from Angular v5 onwards) and how it 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.

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:

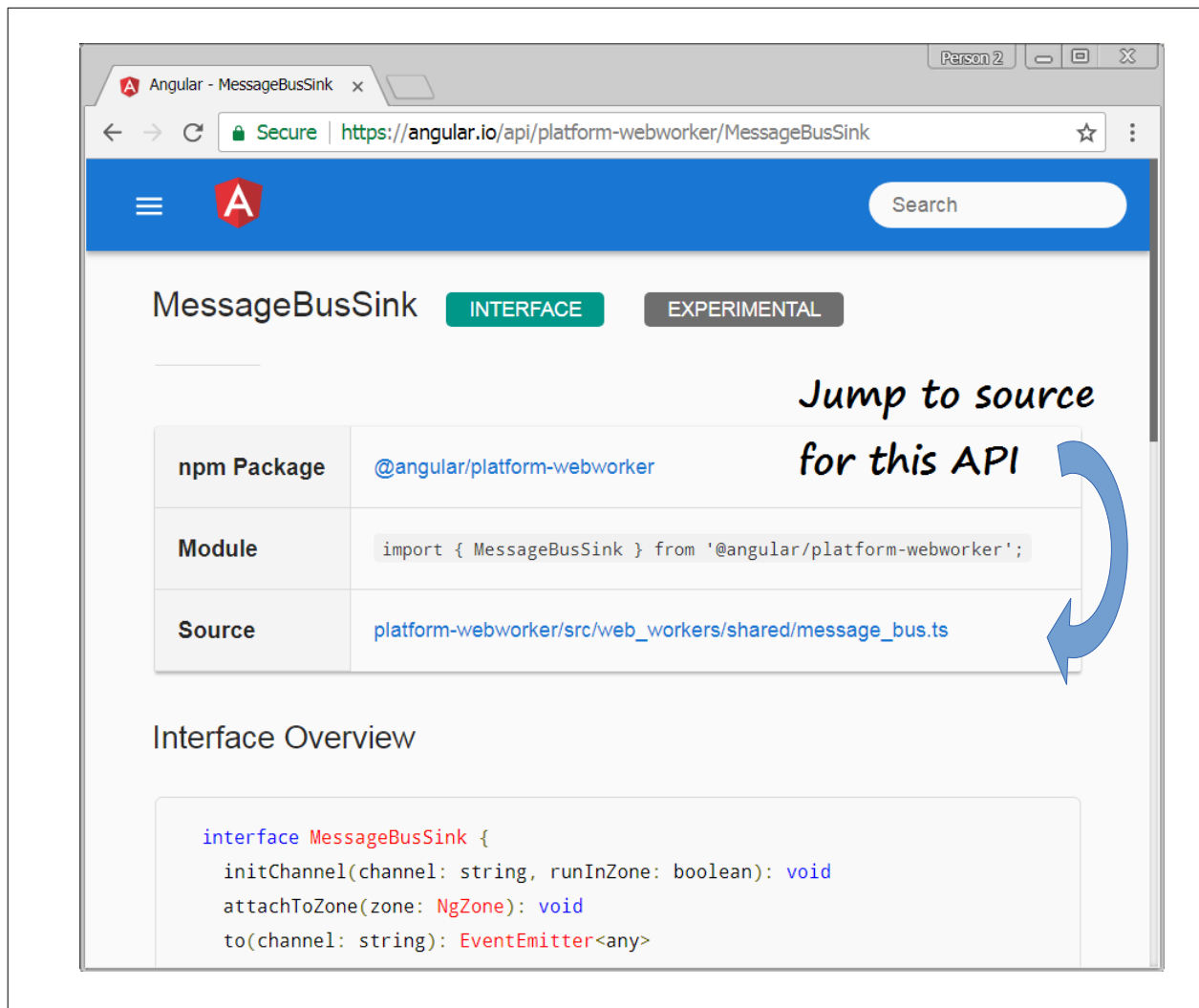
- <https://github.com/angular/angular/blob/master/docs/DEVELOPER.md>

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/api>), the API page for each Angular type has a hyperlink at the top 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).



Keeping up to date

Angular is a fast evolving project. To keep up to date, read the development team's weekly meeting notes, which are available here:

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

The Changelog in Github lists features of new releases:

- <https://github.com/angular/angular/blob/master/CHANGELOG.md>

1: Internals Of Zone.js

This document explores the internals of the Zone.js library and how it is used from within a sample large framework, Angular 12.

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 web worker). 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 that seemingly run concurrently (but really share the same VM event loop). In effect, Zone.js helps you write multithreaded code within a single thread.

When Zone.js is loaded by your app, it monkey-patches certain asynchronous calls (e.g. `setTimeout`, `addEventListener`) to implement zone functionality. Zone.js does this by adding 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 UI framework, like Angular).

Zone.js is mostly transparent to application code. Zone.js runs in the background and for the most part “just works”. For example, Angular uses Zone.js and Angular application code usually runs inside a zone. Tens of thousands of Angular application developers have their code execute within zones without even knowing these exist. More advanced applications can have custom code to make zone API calls if needed and become more actively involved in zone management. Some application developers may wish to take certain steps to move some non-UI performance-critical code outside of the Angular zone – using the [NgZone](#) class).

Project Information

The homepage + root of the source tree for Zone.js are in the main Angular repo at:

- <https://github.com/angular/angular/tree/master/packages/zone.js>

Below we assume you have got the Zone.js source tree downloaded locally under a directory we will call `<ZONE-MASTER>` and any pathnames we use will be relative to that. We also will look at the Angular code that calls Zone.js and we use the `<ANGULAR-MASTER>` as the root of that source tree.

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 about 3 MB.

Using Zone.js

Before looking in detail at how Zone.js itself is implemented, we will first look at how it is used in production, using the example of the very popular Angular project.

The use of Zone.js with Angular is optional, but used by default. Use of zones is mostly a good idea but there are some scenarios (e.g. using Angular Elements to build Web Components) when this is not the case.

To use Zone.js in your applications, you need to load it. Your package.json file will need (if creating a project using Angular CLI, this is added automatically for you):

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

You should load Zone.js after loading core.js (if using that). For example, 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:

```
/*  
 * Zone JS is required by default for Angular itself.  
 */  
import 'zone.js/dist/zone'; // Included with Angular CLI.
```

Angular CLI also generates an angular.json configuration file, with this line that sets up polyfills:

```
"build": {  
  "builder": "@angular-devkit/build-angular:browser",  
  "options": {  
    ..  
    "polyfills": "src/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-MASTER>/dist/zone.js.d.ts](#)

(IMPORTANT: 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 packages directory

(and a similar one under its modules directory and tools directory):

- [<ANGULAR-MASTER>/packages/types.d.ts](#)

and it has the following contents:

```
/// <reference types="hammerjs" />
/// <reference types="jasmine" />
/// <reference types="jasminewd2" />
/// <reference types="node" />
/// <reference types="zone.js" />
/// <reference lib="es2015" />
/// <reference path="./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. The end of the file [<ZONE-MASTER>/blob/master/MODULE.md](#) explains where Angular uses zones:

"Angular uses zone.js to manage async operations and decide when to perform change detection. Thus, in Angular, the following APIs should be patched, otherwise Angular may not work as expected.

- ZoneAwarePromise
- timer
- on_property
- EventTarget
- XHR"

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/api/core/ErrorHandler>

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/api/core/NgZone>

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.

Elsewhere 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>/packages/core/src/zone](#)

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

- [<ANGULAR-MASTER>/packages/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). Both these functions are in `application_ref.ts`. `bootstrapModule<M>` calls the Angular compiler **1** and then calls `bootstrapModuleFactory<M>` **2**.

```
bootstrapModule<M>(
  moduleType: Type<M>, compilerOptions:
  (CompilerOptions&BootstrapOptions) |
  Array<CompilerOptions&BootstrapOptions> = []): Promise<NgModuleRef<M>>
{
  const options = optionsReducer({}, compilerOptions);
1  return compileNgModuleFactory(this.injector, options, moduleType)
    .then(moduleFactory
2           => this.bootstrapModuleFactory(moduleFactory, options));
}
```

It is in `bootstrapModuleFactory` we see how zones are initialized for Angular:

```
bootstrapModuleFactory<M>(moduleFactory: NgModuleFactory<M>, options?:
BootstrapOptions):
  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.
  const ngZoneOption = options ? options.ngZone : undefined;
1  const ngZone = getNgZone(ngZoneOption);
  const providers: StaticProvider[] = [{provide: NgZone, useValue: ngZone}];
  // Attention: Don't use ApplicationRef.run here,
  // as we want to be sure that all possible constructor calls
  // are inside `ngZone.run`!
2  return ngZone.run(() => {
    const ngZoneInjector = Injector.create(
      {providers: providers, parent: this.injector,
       name: moduleFactory.moduleType.name});
    const moduleRef =
      <InternalNgModuleRef<M>>moduleFactory.create(ngZoneInjector);
    const exceptionHandler: ErrorHandler =
3       moduleRef.injector.get(ErrorHandler, null);
    if (!exceptionHandler) {
      throw new Error(
        'No ErrorHandler. Is platform module (BrowserModule) included?');
    }
    moduleRef.onDestroy(() => remove(this._modules, moduleRef));
    ngZone !.runOutsideAngular(
```

```

4    () => ngZone !.onError.subscribe(
        {next: (error: any) =>
            { exceptionHandler.handleError(error); }}});
return _callAndReportToErrorHandler(exceptionHandler, ngZone !,
    () => {
        const initState: ApplicationInitStatus =
            moduleRef.injector.get(ApplicationInitStatus);
        initState.runInitializers();
        return initState.donePromise.then(() => {
5     this._moduleDoBootstrap(moduleRef);
            return moduleRef;
        });
    });
});
}

```

At **1** we see a new `NgZone` being created and at **2** its `run()` method being called, at **3** 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 **4**, we see that error handler being used to configure error handling for the newly created `NgZone`. Finally at **5**, we see the call to the actual bootstrapping.

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

API Model

Zone.js exposes an API for applications to use in the [<ZONE-MASTER>/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`:

```

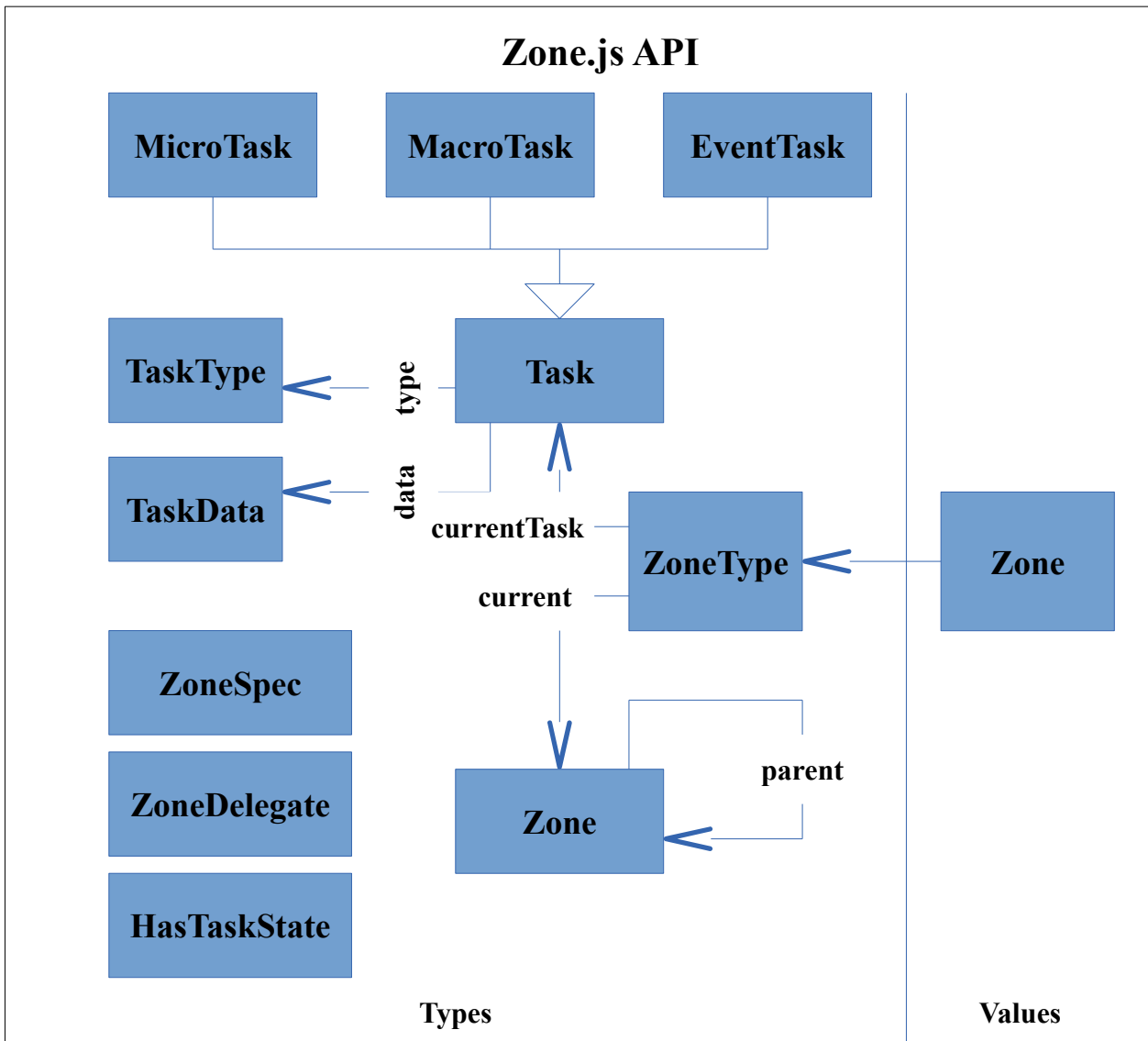
interface ZoneType {
    /**
     * @returns {Zone} Returns the current [Zone]. The only way to change
     * the current zone is by invoking a run() method, which will update the
     * current zone for the duration of the run method callback.
     */
    current: Zone;
    /**
     * @returns {Task} The task associated with the current execution.
     */
    currentTask: Task | null;
    /**
     * Verify that Zone has been correctly patched.
     * Specifically that Promise is zone aware.
     */
    assertZonePatched(): void;
    /**
     * Return the root zone.
     */
    root: Zone;
}

```

```

}
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>/packages/core/src/zone/ng_zone.ts](https://github.com/angular/angular/blob/master/packages/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(`In this configuration Angular requires Zone.js`);
  }
}
```

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 = 'microTask' | 'macroTask' | 'eventTask';
```

`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 `setTimeout`.

```
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;
  state: TaskState;
  source: string;
  invoke: Function;
  callback: Function;
  data?: TaskData;
  scheduleFn?: (task: Task) => void;
  cancelFn?: (task: Task) => void;
  readonly zone: Zone;
  runCount: number;
  cancelScheduleRequest(): void;
}
```

There are three marker interfaces derived from `Task`:

```
interface MicroTask extends Task { type: 'microTask'; }
interface MacroTask extends Task { type: 'macroTask'; }
interface EventTask extends Task { type: 'eventTask'; }
```

The comments for `Task` nicely explains their purpose:

```
* - [MicroTask] queue represents a set of tasks which are executing right
*   after the current stack frame becomes clean and before a VM yield. All
```

```

* [MicroTask]s execute in order of insertion before VM yield and the next
* [MacroTask] is executed.
* - [MacroTask] queue represents a set of tasks which are executed one at a
* time after each VM yield. The queue is ordered by time, and insertions
* can happen in any location.
* - [EventTask] is a set of tasks which can at any time be inserted to the
* end of the [MacroTask] queue. This happens when the event fires.

```

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 forking (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?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
        targetZone: Zone, hasTaskState: HasTaskState) => void;
}

```

The definition of the `Zone` type is:

```

interface Zone {
  parent: Zone | null;
  name: string;
  get(key: string): any;
  getZoneWith(key: string): Zone | null;
  fork(zoneSpec: ZoneSpec): Zone;
  wrap<F extends Function>(callback: F, source: string): F;
  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;
  scheduleTask<T extends Task>(task: T): T;
  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 of 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-MASTER>/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
- doc
- example
- scripts
- tests

The main source is in lib:

- lib

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

Root directory

The root directory contains these markdown documentation files:

- CHANGELOG.md
- DEVELOPER.md
- MODULE.md
- NON-STANDARD-APIS.md
- README.md
- SAMPLE.md
- STANDARD-APIS.md

When we examine [<ZONE-MASTER>/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:

- tslint.json
- tsconfig[-node|-esm-node|esm|].json
- package.json

There are multiple files starting with tsconfig – here are the compilerOptions from the main one:

```
"compilerOptions": {
  "module": "commonjs",
  "target": "es5",
  "noImplicitAny": true,
  "noImplicitReturns": false,
  "noImplicitThis": false,
  "outDir": "build",
  "inlineSourceMap": true,
  "inlineSources": true,
  "declaration": false,
  "noEmitOnError": false,
  "stripInternal": false,
  "strict": true,
  "lib": [
```

```

    "es5",
    "dom",
    "es2015.promise",
    "es2015.symbol",
    "es2015.symbol.wellknown"
  ]
}

```

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": "0.9.0",
  "description": "Zones for JavaScript",
  1 "main": "dist/zone-node.js",
  2 "browser": "dist/zone.js",
  "unpkg": "dist/zone.js",
  "typings": "dist/zone.js.d.ts",
  "files": [
    "lib",
    "dist"
  ],
  "directories": {
    "lib": "lib",
    "test": "test"
  },
}

```

and a list of scripts:

```

"scripts": {
  "changelog": "gulp changelog",
  "ci": "npm run lint && npm run format && npm run promisetest
        && npm run test:single && npm run test-node",
  "closure:test": "scripts/closure/closure_compiler.sh",
  "format": "gulp format:enforce",
  "karma-jasmine": "karma start karma-build-jasmine.conf.js",
  "karma-jasmine:es2015": "karma start karma-build-jasmine.es2015.conf.js",
  "karma-jasmine:phantomjs": "karma start
    karma-build-jasmine-phantomjs.conf.js --single-run",
  "karma-jasmine:single": "karma start karma-build-jasmine.conf.js
    --single-run",
  "karma-jasmine:autoclose": "npm run karma-jasmine:single
    && npm run ws-client",
  "karma-jasmine-phantomjs:autoclose":
    "npm run karma-jasmine:phantomjs && npm run ws-client",
  "lint": "gulp lint",
  "prepublish": "tsc && gulp build",
  "promisetest": "gulp promisetest",
  "promisefinallytest": "mocha promise.finally.spec.js",
  "ws-client": "node ./test/ws-client.js",
  "ws-server": "node ./test/ws-server.js",
  "tsc": "tsc -p .",
  "tsc:w": "tsc -w -p .",
  "tsc:esm2015": "tsc -p tsconfig-esm-2015.json",
  "tslint": "tslint -c tslint.json 'lib/**/*.ts'",
  // many test scripts
  ...
},

```

It has no dependencies:

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

It has many devDependencies.

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 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']
  }
}
```

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 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 and run tests.

All of these tasks result in a call to a local method `generateScript` 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. A number of implementations of Zone

are provided, such as 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. Assuming you have created your Angular project using Angular CLI, it will automatically have set everything up correctly (no manual steps are needed to use Zone.js).

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

- 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-MASTER>/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 (and its sub-directories) contains these scripts:

- grab-blink-idl.sh
- closure/closure_compiler.sh
- closure/closure_flagfile
- sauce/sauce_connect_setup.sh
- sauce/sauce_connect_block.sh

Source Model

The main source for Zone.js is in:

- [<ZONE-MASTER>/lib](#)

It contains a number of sub-directories:

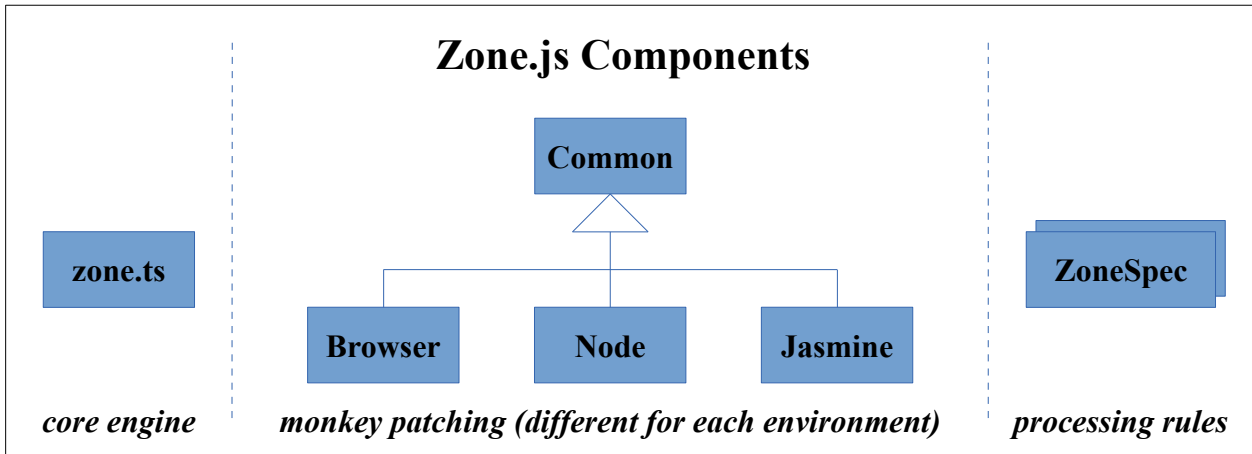
- browser
- closure
- common
- extra
- jasmine
- mix (a mix of browser and node)
- mocha
- node

- rxjs
- testing
- 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 a number of environments supported by Zone.js that need monkey patching, such as the browser, the server (node) and Jasmine and each of these has a sub-directory 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 six 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;
  ...
})
```

The `_ZonePrivate` interface defines private information that is managed per zone:

```
interface _ZonePrivate {
  currentZoneFrame: () => _ZoneFrame;
  symbol: (name: string) => string;
  scheduleMicroTask: (task?: MicroTask) => void;
```

```

onUnhandledError: (error: Error) => void;
microtaskDrainDone: () => void;
showUncaughtError: () => boolean;
patchEventTarget: (global: any, apis: any[], options?: any) => boolean[];
patchOnProperties: (obj: any, properties: string[]|null,
  prototype?: any) => void;
patchThen: (ctro: Function) => void;
setNativePromise: (nativePromise: any) => void;
patchMethod: ..
bindArguments: (args: any[], source: string) => any[];
patchMacroTask:
  (obj: any, funcName: string,
    metaCreator: (self: any, args: any[]) => any) => void;
patchEventPrototype: (_global: any, api: _ZonePrivate) => void;
isIEOrEdge: () => boolean;
ObjectDefineProperty:
  (o: any, p: PropertyKey,
    attributes: PropertyDescriptor&ThisType<any>) => any;
ObjectGetOwnPropertyDescriptor:
  (o: any, p: PropertyKey) => PropertyDescriptor | undefined;
ObjectCreate(o: object|null, properties?
  : PropertyDescriptorMap&ThisType<any>): any;
..
}

```

When using the client API you will not see this, but when debugging through the Zone.js implementation, it will crop up from time to time.

A microtask queue is managed, which requires these variables:

```

let _microTaskQueue: Task[] = [];
let _isDrainingMicrotaskQueue: boolean = false;
let nativeMicroTaskQueuePromise: any;

```

`_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 `_nativeMicroTaskQueuePromise` is used to access a native microtask queue. Which is stored as a global is not set. Two functions manage a microtask queue:

- `scheduleMicroTask`
- `drainMicroTaskQueue`

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<T extends TaskType> implements Task {
  public type: T;
  public source: string;
}

```

```

public invoke: Function;
public callback: Function;
public data: TaskData|undefined;
public scheduleFn: ((task: Task) => void)|undefined;
public cancelFn: ((task: Task) => void)|undefined;
_zone: Zone|null = null;
public runCount: number = 0;
_zoneDelegates: ZoneDelegate[]|null = null;
_state: TaskState = 'notScheduled';

```

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

```

constructor(
  type: T, source: string, callback: Function,
  options: TaskData|undefined,
  scheduleFn: ((task: Task) => void)|undefined,
  cancelFn: ((task: Task) => void)|undefined) {
  this.type = type;
  this.source = source;
  this.data = options;
  this.scheduleFn = scheduleFn;
  this.cancelFn = cancelFn;
  this.callback = callback;
  const self = this;
  // TODO: @JiaLiPassion options should have interface
  if (type === eventTask && options && (options as any).useG) {
    this.invoke = ZoneTask.invokeTask;
  } else {
    this.invoke = function() {
      return ZoneTask.invokeTask.call(
        global, self, this, <any>arguments);
    };
  }
}

```

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`.

```

static invokeTask(task: any, target: any, args: any): any {
  if (!task) {
    task = this;
  }
  _numberOfNestedTaskFrames++;
  try {
    task.runCount++;
    return task.zone.runTask(task, target, args);
  } 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.toString();
    } else {
      return Object.prototype.toString.call(this);
    }
  }
}

```

`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];
        try {
          task.zone.runTask(task, null, null);
        } catch (error) {
          _api.onUnhandledError(error);
        }
      }
    }
    _api.microtaskDrainDone();
    _isDrainingMicrotaskQueue = false;
  }
}

```

The `_microTaskQueue` gets populated via a call to `scheduleMicroTask`:

```

function scheduleMicroTask(task?: MicroTask) {
  // 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 (!nativeMicroTaskQueuePromise) {
      if (global[symbolPromise]) {
        nativeMicroTaskQueuePromise = global[symbolPromise].resolve(0);
      }
    }
    if (nativeMicroTaskQueuePromise) {
      let nativeThen = nativeMicroTaskQueuePromise[symbolThen];
      if (!nativeThen) {
        // native Promise is not patchable, we need to use `then` directly
        // issue 1078
        nativeThen = nativeMicroTaskQueuePromise['then'];
      }
      nativeThen.call(nativeMicroTaskQueuePromise, drainMicroTaskQueue);
    } else {
      global[symbolSetTimeout](drainMicroTaskQueue, 0);
    }
  }
  task && _microTaskQueue.push(task);
}

```

If needed (not running in a task), this 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 more 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.

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.

```
private _interceptDlgt: ZoneDelegate|null;
private _interceptZS: ZoneSpec|null;
private _interceptCurrZone: Zone|null;
```

`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|null;
```

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|null, zoneSpec: ZoneSpec|null
){
  this.zone = zone;
  this._parentDelegate = parentDelegate;

  this._forkZS = zoneSpec
  && (zoneSpec && zoneSpec.onFork ? zoneSpec : parentDelegate!._forkZS);
  this._forkDlgt = zoneSpec
  && (zoneSpec.onFork ? parentDelegate : parentDelegate!._forkDlgt);
  this._forkCurrZone = zoneSpec
  && (zoneSpec.onFork ? this.zone : parentDelegate!.zone);
```

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._invokeCurrZone!,
     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 {
  let returnTask: ZoneTask<any> = task as ZoneTask<any>;
  if (this._scheduleTaskZS) {
1    if (this._hasTaskZS) {
        returnTask._zoneDelegates!.push(this._hasTaskDlgtOwner!);
      }
    returnTask = this._scheduleTaskZS.onScheduleTask!
      (this._scheduleTaskDlgt!,
       this._scheduleTaskCurrZone!, targetZone, task)
      as ZoneTask<any>;
    if (!returnTask) returnTask = task as ZoneTask<any>;
  } else {
2    if (task.scheduleFn) {
3      task.scheduleFn(task);
    } else if (task.type == microTask) {
4      scheduleMicroTask(<MicroTask>task);
    } else {
      throw new Error('Task is missing scheduleFn.');
    }
  }
  return returnTask;
}
```

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 `_currentZoneFrame` is initialized to the root zone and `_currentTask` to null:

```
let _currentZoneFrame: _ZoneFrame
  = {parent: null, zone: new Zone(null, null)};
let _currentTask: Task|null = null;
let _numberOfNestedTaskFrames = 0;
```

2: Tsickle

Overview

Tsickle is a small utility that carries out transformations on TypeScript code to make it suitable as input for the Closure Compiler, which in turn is used to generate highly optimized JavaScript code.

The home for the Tsickle project is:

- <https://github.com/angular/tsickle>

which introduces Tsickle as:

“Tsickle converts TypeScript code into a form acceptable to the Closure Compiler. This allows using TypeScript to transpile your sources, and then using Closure Compiler to bundle and optimize them, while taking advantage of type information in Closure Compiler.”

To learn more about Closure, visit:

- <https://github.com/google/closure-compiler>

Tsickle is used Angular but can also be used by non-Angular projects.

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:

Project Setup

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": "built/src/tsickle.js",
"bin": "built/src/main.js",
```

The gulpfile.js file contains the following Gulp tasks:

- gulp format
- gulp test.check-format (formatting tests)
- gulp test.check-lint (run tslint)

The src Sub-Directory

The src sub-directory contains the following source files:

- class_decorator_downlevel_transformer.ts
- cli_support.ts
- closure_externs.js
- decorator-annotator.ts
- decorators.ts
- es5processor.ts
- fileoverview_comment_transformer.ts
- jsdoc.ts
- main.ts
- modules_manifest.ts
- rewriter.ts
- source_map_utils.ts
- transformer_sourcemap.ts
- transformer_util.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 const {settings, tscArgs} = loadSettingsFromArgs(args);
2 const config = loadTscConfig(tscArgs);
  if (config.errors.length) {
    console.error(tsickle.formatDiagnostics(config.errors));
    return 1;
  }

  if (config.options.module !== ts.ModuleKind.CommonJS) {
    // This is not an upstream TypeScript diagnostic, therefore it does not
    // go through the diagnostics array mechanism.
    console.error(
      'tsickle converts TypeScript modules to Closure modules via
        'CommonJS internally. Set tsconfig.js "module": "commonjs"');
  }
}
```

```

    return 1;
  }

  // Run tsickle+TSC to convert inputs to Closure JS files.
  3 const result = toClosureJS(
    config.options, config.fileNames, settings,
    (filePath: string, contents: string) => {
      mkdirp.sync(path.dirname(filePath));
      4 fs.writeFileSync(filePath, contents, {encoding: 'utf-8'});
    });
  if (result.diagnostics.length) {
    console.error(tsickle.formatDiagnostics(result.diagnostics));
    return 1;
  }

  5 if (settings.externsPath) {
    mkdirp.sync(path.dirname(settings.externsPath));
    fs.writeFileSync(settings.externsPath,
      tsickle.getGeneratedExterns(result.externs));
  }
  return 0;
}

// CLI entry point
if (require.main === module) {
  process.exit(main(process.argv.splice(2)));
}

```

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, if so configured.

```

export function toClosureJS(
  options: ts.CompilerOptions, fileNames: string[], settings: Settings,
  writeFile?: ts.WriteFileCallback): tsickle.EmitResult {
  1 const compilerHost = ts.createCompilerHost(options);
  2 const program = ts.createProgram(fileNames, options, compilerHost);

  3 const transformerHost: tsickle.TsickleHost = {
    shouldSkipTsickleProcessing: (fileName: string) => {
      return fileNames.indexOf(fileName) === -1;
    },
    shouldIgnoreWarningsForPath: (fileName: string) => false,
    pathToModuleName: cliSupport.pathToModuleName,
    fileNameToModuleId: (fileName) => fileName,
    es5Mode: true, googmodule: true, prelude: '',
    transformDecorators: true, transformTypesToClosure: true,
    typeBlackListPaths: new Set(), untyped: false,
    logWarning: (warning) =>
      console.error(tsickle.formatDiagnostics([warning])),

```

```

};
const diagnostics = ts.getPreEmitDiagnostics(program);
if (diagnostics.length > 0) {
  return {
    diagnostics,
    modulesManifest: new ModulesManifest(),
    externs: {},
    emitSkipped: true,
    emittedFiles: [],
  };
}
4 return tsickle.emitWithTsickle(
  program, transformerHost, compilerHost, options, undefined, writeFile);
}

```

It first creates **1** a compiler host based on the supplied options, then **2** it uses TypeScript's `createProgram` method with the original program source to ensure it is syntactically correct and any error messages refer the original source, not the modified source. Then it creates **3** a `tsickle.TsickleHost` instance which it passes **4** to `tsickle.emitWithTsickle()`.

The `annotate` function is a simple function:

```

export function annotate(
  typeChecker: ts.TypeChecker, file: ts.SourceFile, host: AnnotatorHost,
  tsHost?: ts.ModuleResolutionHost, tsOpts?: ts.CompilerOptions,
  sourceMapper?: SourceMapper): {output: string, diagnostics:
ts.Diagnostic[]} {
  return new Annotator(typeChecker, file, host, tsHost, tsOpts,
sourceMapper).annotate();
}

```

Classes called `rewriters` are used to rewrite the source. The `rewriter.ts` file has the `Rewriter` abstract class. An important method is `maybeProcess()`.

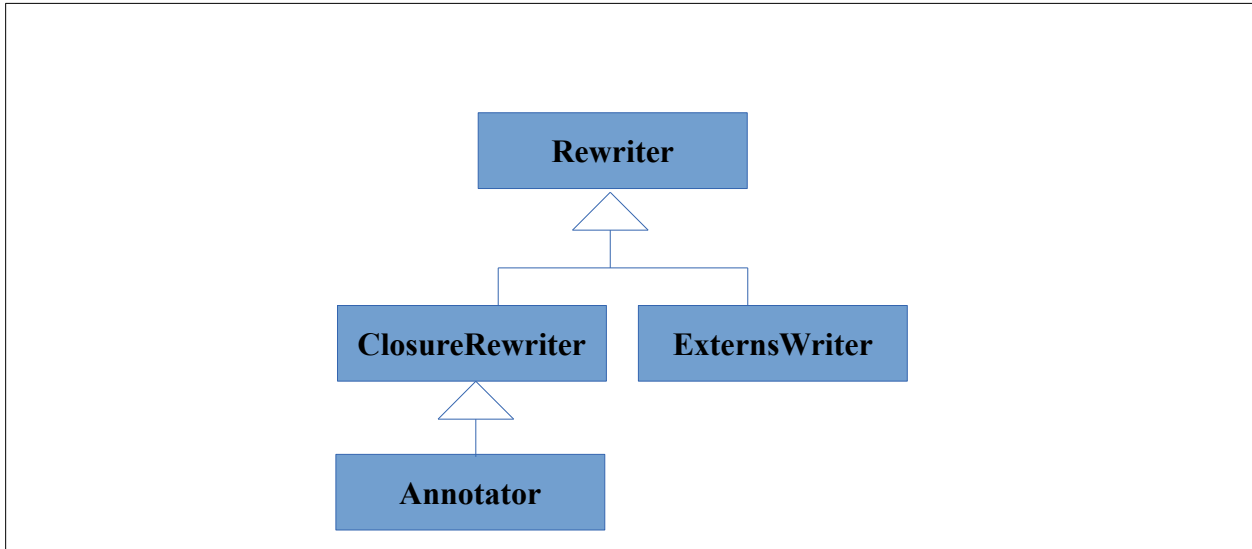
```

/**
 * 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 {
  private output: string[] = [];
  /** Errors found while examining the code. */
  protected diagnostics: ts.Diagnostic[] = [];
  /** Current position in the output. */
  private position: SourcePosition = {line: 0, column: 0, position: 0};
  /**
   * The current level of recursion through TypeScript Nodes. Used in
   formatting internal debug
   * print statements.
   */
  private indent = 0;
  /**
   * Skip emitting any code before the given offset.
   * E.g. used to avoid emitting @fileoverview
   * comments twice.
   */
  private skipCommentsUpToOffset = -1;

```

```
constructor(public file: ts.SourceFile,  
            private sourceMapper: SourceMapper = NOOP_SOURCE_MAPPER) { }
```

`tsickle.ts` has some classes that derive from `Rewriter`, according to this hierarchy:



`Annotator.maybeProcess()` is where the actual rewriting occurs.

3: TS-API-Guardian

Overview

Ts-api-guardian is a small tool that tracks a package's public API.

Usage

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](#)

you will see it has these two lines:

```
gulp.task('public-api:enforce', loadTask('public-api', 'enforce'));
gulp.task('public-api:update', ['build.sh'],
  loadTask('public-api', 'update'));
```

and when we look at:

- [<ANGULAR-MASTER>/tools/gulp-tasks/public-api.js](#)

we see two tasks named 'public-api:enforce' and 'public-api:update' and In here we see how ts-api-guardian is used, to **1** ensure it has not been unexpectedly changed and to **2** generate a "golden file" representing the API:

```
// Enforce that the public API matches the golden files
// Note that these two commands work on built d.ts files
// instead of the source
1 enforce: (gulp) => (done) => {
  const platformScriptPath = require('./platform-script-path');
  const childProcess = require('child_process');
  const path = require('path');

  childProcess
    .spawn(path.join(__dirname,
      platformScriptPath(`../../node_modules/.bin/ts-api-guardian`)),
      ['--verifyDir', path.normalize(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
2 update: (gulp) => (done) => {
  const platformScriptPath = require('./platform-script-path');
```



```
const childProcess = require('child_process');
const path = require('path');

childProcess
  .spawn(
    path.join(__dirname, platformScriptPath(
      `../../node_modules/.bin/ts-api-guardian`)),
    ['--outDir', path.normalize(publicApiDir)].concat(publicApiArgs),
    {stdio: 'inherit'})
  .on('close', done);
}
```

Source Tree

The root directory contains these files:

- gulpfile.js
- package.json
- tsconfig.json
- tsd.json

and these top-level directories:

- bin
- lib
- test

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": "2.0.10"
},
```

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

The bin directory has one file, ts-api-guardian, which just has these lines:

```
#!/usr/bin/env node
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 <identifier>
                        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 them. If they 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)`

4: RxJS 6

Introduction

RxJS is a wonderful framework to handle observable streams of items. We see its use in Angular as event emitters, HTTP responses and services. To become a good Angular developer you first have to become a good RxJS developer. It is that critical to the correct programming of Angular. RxJS is also the base technology for NgRx, the very popular state cache engine for client apps.

An observable is a dual of an enumerable. With an enumerable, your code would call `.next()` repeatedly to get the next item in a sequence. In contrast, with an observable, you supply one method to handle new data items, and optionally a method to handle errors and another to handle the completion event (successful end of items). The internals of the observable will call your code when it has data items to deliver to you. So you could imagine a HTTP request being sent to the server, and a blob of data coming back in multiple packets. As more items come in, the observable will call the handler in your code to process them.

Project Information

RxJS has gone through many iterations and recently it has been re-written in TypeScript.

The project home page is: <http://reactivex.io/rxjs>

To get started with the source tree, visit the repo on Github:

- <https://github.com/ReactiveX/rxjs>

API Model

In the links below, we assume `<RXJS>` refers to the root directory of your rRxJS source installation. The links below actually link to the relevant document on github.

The main exports are in:

- <RXJS>/src/index.ts

with a large range of operators in:

- <RXJS>/src/operators/index.ts

Application developers who wish to use RxJS will mostly import what they need from these two files. There are sub-directories inside `src/internal` and `src/operators/internal` and as the name suggests, external developers should not be directly importing (or in any way depending on) these, as they may (will) change from version to version.

Developers with old code base that rely on older versions of RxJS should at some stage upgrade to RxJS 6 or later. But as an interim measure, a compatibility layer is

available to help old code run against RxJS 6. The compat package is at:

- [<RXJS>/compat](#)

If you have a large codebase it is not always convenient to immediately upgrade to RxJS 6, so a compat package can be very useful. You should consider use fo compat as a temporary measure. Over time you at first should and later must upgrade, as it is expected the compat package will disappear at some point in future. We do not discuss the compat package further here.

What [<RXJS>/src/index.ts](#) exports can be subdivided into the following categories:

- Observables
- Subjects (these are both observers and observables)
- Schedulers
- Subscription
- Notification
- Utils
- Error Types
- Static observable creation functions
- Two constants (EMPTY and NEVER)
- config

So where is observer exported? Hmm ... There is also one `export *`:

```
export * from './internal/types';
```

and if we look at:

- [<RXJS>/src/types.ts](#)

we see `Observer` defined as an interface in there, along with definitions of operator interfaces, subscription interfaces, observable interfaces and a range of other observer interfaces – we will shortly explore them all.

Source Tree Model

The root directory contains these sub-directories:

- .git
- github
- compat
- doc
- integration
- legacy-reexport
- migrations
- node-tests
- perf
- spec
- src
- tools
- tsconfig

Files in root directory

There are these JSON files:

- .dependency-cruiser.json
- esdoc.json
- package.json
- tsconfig.base.json
- tsconfig.json
- tsconfig.vscode.json
- tslint.json

There are these .js files:

- .make-compat0package.js
- .make-helpers.js
- .make-packages.js
- .markdown-doctest-setup.js
- dangerfile.js
- index.js
- protractor.conf.js
- wallaby.js

There are these . files:

- .editorconfig
- .eslintrc
- .gitignore
- .gitattributes
- .jshintrc
- .travis.yml
- .npmignore

There are these markdown documentation files:

- CHANGELOG.md
- CODE_OF_CONDUCT.md
- CONTRIBUTING.md
- MIGRATION.md

There are two shell scripts:

- publish.sh
- publish_docs.sh

There are these other files:

- MAINTAINERS
- LICENSE.txt
- appveyor.yml

5: The Core Package

Overview

Core is the foundational Angular package upon which all other packages are based. It supplies a wide range of functionality, 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`.

Core Public API

The `index.ts` file in Core's root directory just exports the contents of the `public-api` file, which in turn exports the contents of `src/core.ts`:

- [<ANGULAR-MASTER>/packages/core/src/core.ts](https://github.com/angular/angular/blob/master/packages/core/src/core.ts)

which is where Core's exported API is defined:

```
export * from './metadata';
export * from './version';
export {TypeDecorator} from './util/decorators';
export * from './di';
export {createPlatform, assertPlatform, destroyPlatform, getPlatform,
  PlatformRef, ApplicationRef, createPlatformFactory, NgProbeToken}
  from './application_ref';
export {enableProdMode, isDevMode} from './util/is_dev_mode';
export {APP_ID, PACKAGE_ROOT_URL, PLATFORM_INITIALIZER, PLATFORM_ID,
  APP_BOOTSTRAP_LISTENER} from './application_tokens';
export {APP_INITIALIZER, ApplicationInitStatus} from './application_init';
export * from './zone';
export * from './render';
export * from './linker';
export {DebugElement, DebugEventListener, DebugNode, asNativeElements,
  getDebugNode, Predicate} 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,
  MissingTranslationStrategy} from './i18n/tokens';
export {ApplicationModule} from './application_module';
export {wtfCreateScope, wtfLeave, wtfStartTimeRange, wtfEndTimeRange,
  WtfScopeFn} from './profile/profile';
export {AbstractType, Type} from './interface/type';
export {EventEmitter} from './event_emitter';
export {ErrorHandler} from './error_handler';
export * from './core_private_export';
export * from './core_render3_private_export';
export {Sanitizer, SecurityContext} from './sanitization/security';
export * from './codegen_private_exports';
```

If you are writing a normal Angular application, to use the Core package you will import Core's `index.ts`. There are two additional files with exports, which are used

internally within Angular, to allow other Angular packages import additional types from the Core package (that are not in the normal `index.ts`). These two files are named `private_exports` and are located in Core's `src` directory (whereas `index.ts` is located in Core's root directory).

The first of these is:

- [<ANGULAR-MASTER>/packages/core/src/core_private_export.ts](#)

and has these exports:

```
export {ALLOW_MULTIPLE_PLATFORMS as eALLOW_MULTIPLE_PLATFORMS}
  from './application_ref';
export {APP_ID_RANDOM_PROVIDER as eAPP_ID_RANDOM_PROVIDER}
  from './application_tokens';
export {defaultIterableDiffers as edefaultIterableDiffers,
defaultKeyValueDiffers as edefaultKeyValueDiffers} from
  './change_detection/change_detection';
export {devModeEqual as edevModeEqual}
  from './change_detection/change_detection_util';
export {isListLikeIterable as eisListLikeIterable}
  from './change_detection/change_detection_util';
export {ChangeDetectorStatus as eChangeDetectorStatus,
  isDefaultChangeDetectionStrategy as eisDefaultChangeDetectionStrategy}
  from './change_detection/constants';
export {Console as eConsole} from './console';
export {inject, setCurrentInjector as esetCurrentInjector, eeinject}
  from './di/injector_compatibility';
export {getInjectableDef as egetInjectableDef, eeInjectableDef,
  eeInjectorDef} from './di/interface/defs';
export {APP_ROOT as eAPP_ROOT} from './di/scope';
export {ivyEnabled as eivyEnabled} from './ivy_switch';
export {ComponentFactory as eComponentFactory}
  from './linker/component_factory';
export {CodegenComponentFactoryResolver as eCodegenComponentFactoryResolver}
  from './linker/component_factory_resolver';
export {clearResolutionOfComponentResourcesQueue as
  eclearResolutionOfComponentResourcesQueue, resolveComponentResources
  as eresolveComponentResources} from './metadata/resource_loading';
export {ReflectionCapabilities as eReflectionCapabilities}
  from './reflection/reflection_capabilities';
export {GetterFn as eGetterFn, MethodFn as eMethodFn, SetterFn as eSetterFn}
  from './reflection/types';
export {DirectRenderer as eDirectRenderer, RenderDebugInfo as
  eRenderDebugInfo} from './render/api';
export {_sanitizeHtml as e_sanitizeHtml}
  from './sanitization/html_sanitizer';
export {_sanitizeStyle as e_sanitizeStyle}
  from './sanitization/style_sanitizer';
export {_sanitizeUrl as e_sanitizeUrl} from './sanitization/url_sanitizer';
export {global as eglobal} from './util/global';

export {looseIdentical as elooseIdentical,} from './util/comparison';
export {stringify as estringify} from './util/stringify';
export {makeDecorator as emakeDecorator} from './util/decorators';
export {isObservable as eisObservable, isPromise as eisPromise}
  from './util/lang';
export {clearOverrides as eclearOverrides, initServicesIfNeeded as
  einitServicesIfNeeded, overrideComponentView as eoverrideComponentView,
```

```

overrideProvider as eoverrideProvider} from './view/index';
export {NOT_FOUND_CHECK_ONLY_ELEMENT_INJECTOR as
eNOT_FOUND_CHECK_ONLY_ELEMENT_INJECTOR} from './view/provider';
export {getLocalePluralCase as egetLocalePluralCase, findLocaleData as
efindLocaleData} from './i18n/locale_data_api';
export {LOCALE_DATA as eLOCALE_DATA, LocaleDataIndex as eLocaleDataIndex}
from './i18n/locale_data';

```

We observe that the exports are being renamed with the Greek Theta symbol (looks like a 'o' with a horizontal line through it) – all are exported as:

```
export X as eX;
```

For an explanation see here -

<https://stackoverflow.com/questions/45466017/%C9%B5-theta-like-symbol-in-angular-2-source-code>

“The letter θ is used by the Angular team to indicate that some method is private to the framework and must not be called directly by the user, as the API for these method is not guaranteed to stay stable between Angular versions (in fact, I would say it's almost guaranteed to break).”

The second private exports file is:

- [<ANGULAR-MASTER>/packages/core/src/codegen_private_exports.ts](https://github.com/angular/angular/blob/master/packages/core/src/codegen_private_exports.ts)

and has these exports:

```

export {CodegenComponentFactoryResolver as eCodegenComponentFactoryResolver}
  from './linker/component_factory_resolver';
export {registerModuleFactory as eregisterModuleFactory}
  from './linker/ng_module_factory_registration';
export {ArgumentType as eArgumentType, BindingFlags as eBindingFlags,
  DepFlags as eDepFlags, EMPTY_ARRAY as eEMPTY_ARRAY,
  EMPTY_MAP as eEMPTY_MAP, NodeFlags as eNodeFlags,
  QueryBindingType as eQueryBindingType,
  QueryValueType as eQueryValueType,
  ViewDefinition as eViewDefinition,
  ViewFlags as eViewFlags,
  anchorDef as eand
  createComponentFactory as eccf,
  createNgModuleFactory as ecmf,
  createRendererType2 as ecrt,
  directiveDef as edid, elementDef as eeld,
  getComponentViewDefinitionFactory as egetComponentViewDefinitionFactory,
  inlineInterpolate as einlineInterpolate,
  interpolate as einterpolate, moduleDef as emod,
  moduleProvideDef as empd, ngContentDef as encd,
  nodeValue as enov, pipeDef as epid,
  providerDef as eprd, pureArrayDef as epad,
  pureObjectDef as epod, purePipeDef as eppd,
  queryDef as equd, textDef as eted, unwrapValue as eunv,
  viewDef as evid} from './view/index';

```

Source Tree Layout

The Core source tree is at:

- [<ANGULAR-MASTER>/packages/core](https://github.com/angular/angular/blob/master/packages/core)

The source tree for the Core package contains these directories:

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

and these files:

- BUILD.bazel
- index.ts
- package.json
- public_api.ts
- rollup.config.js
- tsconfig-build.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
- di.ts
- metadata.ts
- linker.ts
- render.ts
- zone.ts

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

```
export {RenderComponentType, Renderer, Renderer2, RendererFactory2,
        RendererStyleFlags2, RendererType2, 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 {
    // tslint:disable-next-line:no-console
    console.log(message);
  }
  // Note: for reporting errors use `DOM.logError()`
  // as it is platform specific
}
```

```

warn(message: string): void {
  // tslint:disable-next-line:no-console
  console.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 the 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: StaticProvider[] = [
  // Set a default platform name for platforms that don't set it explicitly.
  {provide: PLATFORM_ID, useValue: 'unknown'},
  {provide: PlatformRef, deps: [Injector]},
  {provide: TestabilityRegistry, deps: []},
  {provide: Console, deps: []},
];

```

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 = 0,
  HTML = 1,
  STYLE = 2,
  SCRIPT = 3,
  URL = 4,
  RESOURCE_URL = 5,
}

```

and the `Sanitizer` abstract class:

```

/**
 * Sanitizer is used by the views to sanitize potentially dangerous values.
 *
 * @stable
 */
export abstract class Sanitizer {
  abstract sanitize(context: SecurityContext,
                  value: {}|string|null) : string|null;
}

```

They are used in [<ANGULAR-MASTER>/packages/compiler/src/schema](#) and [<ANGULAR-MASTER>/packages/platform-browser/src/security](#) to enforce security restrictions on potentially dangerous constructs.

The third category of source files directly in the `src` directory 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 (50-70 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:

```
// A function that will be executed when a platform is initialized.
export const PLATFORM_INITIALIZER =
  new InjectionToken<Array<() => void>>('Platform Initializer');
```

An example usage within Angular is in platform-browser:

- [<ANGULAR-MASTER>/packages/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: StaticProvider[] = [
  {provide: PLATFORM_ID, useValue: PLATFORM_BROWSER_ID},
  {provide: PLATFORM_INITIALIZER, useValue: initDomAdapter, multi: true},
  {provide: PlatformLocation, useClass: BrowserPlatformLocation,
    deps: [DOCUMENT]},
  {provide: DOCUMENT, useFactory: _document, deps: []},
];
```

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?: StaticProvider[]) =>
  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.

```
export const APP_ID = new InjectionToken<string>('AppId');
```

A default implementation is supplied, that uses `Math.random` and this is then used to create an APP_ID_RANDOM_PROVIDER:

```
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:

```

// A function that will be executed when an application is initialized.
export const APP_INITIALIZER =
  new InjectionToken<Array<() => void>>('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:

```

// A class that reflects the state of running {@link APP_INITIALIZER}s.
@Injectable()
export class ApplicationInitStatus {
  private resolve: Function;
  private reject: Function;
  private initialized = false;
  public readonly donePromise: Promise<any>;
  public readonly done = false;

  constructor(@Inject(APP_INITIALIZER)
    @Optional() private appInits: (() => any)[]) {
    this.donePromise = new Promise((res, rej) => {
      this.resolve = res;
      this.reject = rej;
    });
  }
}

```

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,
    ApplicationInitStatus,
    Compiler,
    APP_ID_RANDOM_PROVIDER,
    {provide: IterableDiffers, useFactory: _iterableDiffersFactory},
    {provide: KeyValueDiffers, useFactory: _keyValueDiffersFactory},
    {
      provide: LOCALE_ID,
      useFactory: _localeFactory,
      deps: [[new Inject(LOCALE_ID), new Optional(), new SkipSelf()]]
    },
  ],
})
export class ApplicationModule {
  // Inject ApplicationRef to make it eager...
  constructor(appRef: ApplicationRef) {}
}

```

Providers are supplied to Angular's dependency injection system. 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.

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   parentPlatformFactory:
      ((extraProviders?: StaticProvider[]) => PlatformRef) | null,
2   name: string,
3   providers: StaticProvider[] = [])
4     : (extraProviders?: StaticProvider[]) => PlatformRef {
  const marker = new InjectionToken(`Platform: ${name}`);
5  return (extraProviders: StaticProvider[] = []) => {
    let platform = getPlatform();
    if (!platform || platform.injector.get(ALLOW_MULTIPLE_PLATFORMS, false)) {
      if (parentPlatformFactory) {
        parentPlatformFactory(
          providers.concat(extraProviders).concat(
            {provide: marker, useValue: true}));
      } else {
6        createPlatform(Injector.create(
          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:

```
@Injectable()
export class PlatformRef {
  private _modules: NgModuleRef<any>[] = [];
  private _destroyListeners: Function[] = [];
  private _destroyed: boolean = false;

  /** @internal */
  constructor(private _injector: Injector) {}
```

```

bootstrapModuleFactory<M>(
  moduleFactory: NgModuleFactory<M>,
  options?: BootstrapOptions): Promise<NgModuleRef<M>> { }

bootstrapModule<M>(
  moduleType: Type<M>,
  compilerOptions: (CompilerOptions&BootstrapOptions) | Array<CompilerOptions&BootstrapOptions>=[]
) : Promise<NgModuleRef<M>> { }

private _moduleDoBootstrap(moduleRef: InternalNgModuleRef<any>): void { }

onDestroy(callback: () => void): void { }

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

destroy() { }

get destroyed() { return this._destroyed; }
}

```

A platform represents the hosting environment within which one or more applications execute. Different platforms are supported (e.g. browser UI, web worker, 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.

The supplied implementation of `PlatformRef` manages the root injector passed in to the constructor, an array of `NgModuleRefs` 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.

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()`.

```

bootstrapModule<M>(
  moduleType: Type<M>,
  compilerOptions: (CompilerOptions&BootstrapOptions) |
  Array<CompilerOptions&BootstrapOptions> = []): Promise<NgModuleRef<M>> {

const compilerFactory: CompilerFactory = this.injector.get(CompilerFactory);
const options = optionsReducer({}, compilerOptions);
const compiler = compilerFactory.createCompiler([options]);

```



```

1 return compiler.compileModuleAsync(moduleType)
2 .then((moduleFactory)=>this.bootstrapModuleFactory(moduleFactory,
options));
}

```

`bootstrapModule` first calls **1** the template compiler and then **2** calls `bootstrapModuleFactory`, so after the extra runtime compilation step, both bootstrapping approaches follow the same code path.

When examining zones we already looked at the use of zones within `bootstrapModuleFactory` d- other important code there includes the construction of `moduleRef` **1** and the call to **2** `_moduleDoBootstrap(moduleRef)`:

```

1 const moduleRef =
<InternalNgModuleRef<M>>moduleFactory.create(ngZoneInjector);

return _callAndReportToErrorHandler(exceptionHandler, ngZone !, () => {
  const initStatus: ApplicationInitStatus =
moduleRef.injector.get(ApplicationInitStatus);
  initStatus.runInitializers();
  return initStatus.donePromise.then(() => {
2    this._moduleDoBootstrap(moduleRef);
    return moduleRef;
  });
});

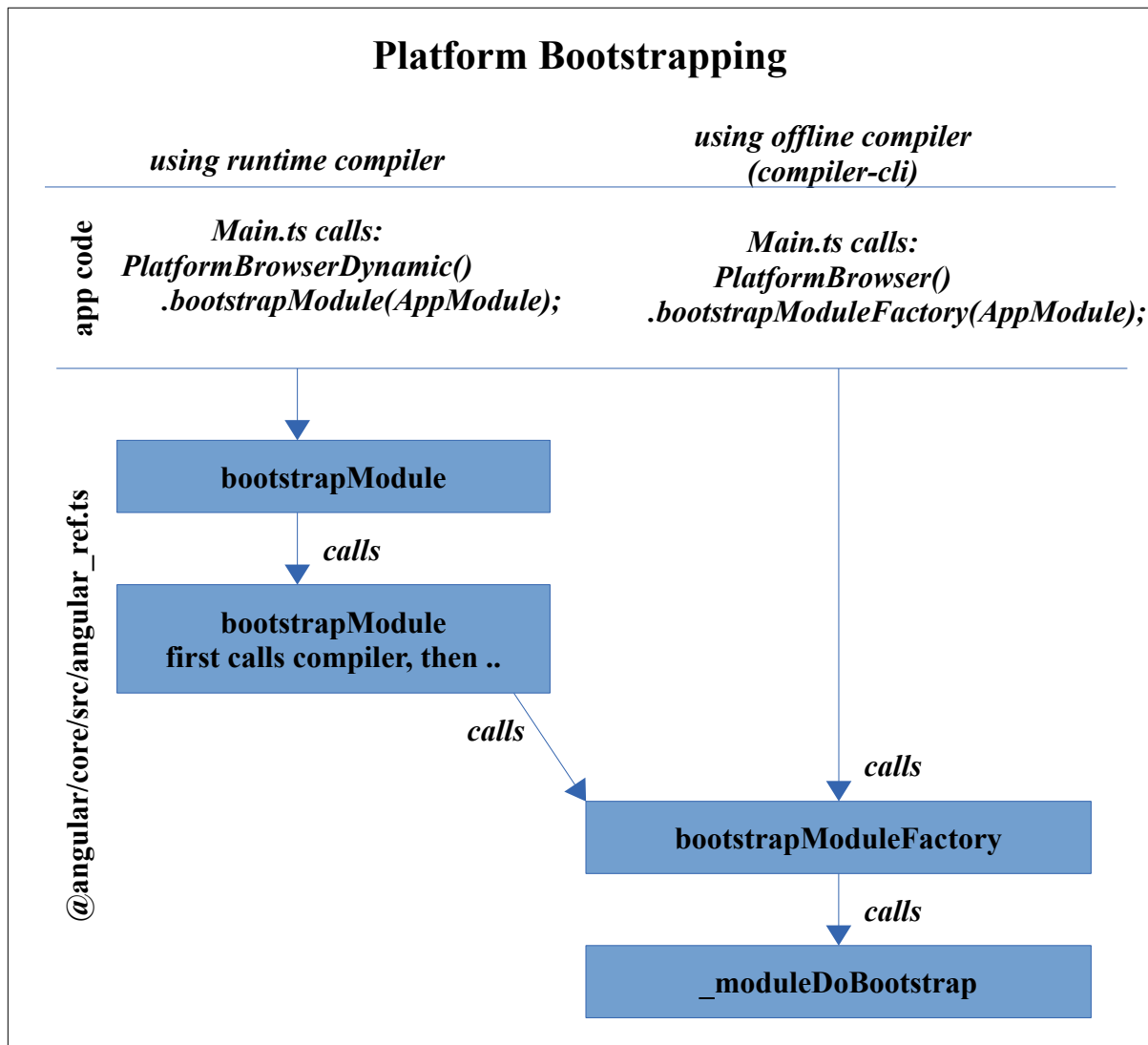
```

It is in `_moduleDoBootstrap` that we see the actual bootstrapping taking place:

```

private _moduleDoBootstrap(moduleRef: InternalNgModuleRef<any>): void {
  const appRef = moduleRef.injector.get(ApplicationRef) as ApplicationRef;
  if (moduleRef._bootstrapComponents.length > 0) {
    moduleRef._bootstrapComponents.forEach(f => appRef.bootstrap(f));
  } 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.`);
  }
  this._modules.push(moduleRef);
}

```



In addition to bootstrapping functionality, 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 and then calls the initializers:

```
/**
 * Creates a platform.
 * Platforms have to be eagerly created via this function.
 */
export function createPlatform(injector: Injector): PlatformRef {
  if (_platform && !_platform.destroyed &&
    !_platform.injector.get(ALLOW_MULTIPLE_PLATFORMS, false)) {
    throw new Error(
      'There can be only one platform.
      Destroy the previous one to create a new one.');
```

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|null {
  return _platform && !_platform.destroyed ? _platform : null;
}
```

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

```
/**
 * Checks that there currently is a platform which contains the
 * given token as a provider.
 */
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 calls the `destroy` method for the platform:

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

The run mode specifies whether the platform is in 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.');
```

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:

```

@Injectables()
export class ApplicationRef {
  public readonly componentTypes: Type<any>[] = [];
  public readonly components: ComponentRef<any>[] = [];
  public readonly isStable: Observable<boolean>;
  bootstrap<C>(componentOrFactory: ComponentFactory<C>|Type<C>,
               rootSelectorOrNode?: string|any): {}

  tick(): void { }
  attachView(viewRef: ViewRef): void { }
  detachView(viewRef: ViewRef): void { }
  get viewCount() { return this._views.length; }
}

```

Its main method is `bootstrap()`, which is a generic method with a 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.

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

```

static _tickScope: WtfScopeFn = wtfCreateScope('ApplicationRef#tick()');
private _bootstrapListeners: ((compRef: ComponentRef<any>) => void)[] = [];
private _views: InternalViewRef[] = [];
private _runningTick: boolean = false;
private _enforceNoNewChanges: boolean = false;
private _stable = true;

```

Its constructor shows what it needs from an injector and sets up the observables (code here is abbreviated):

```

constructor(
  private _zone: NgZone,
  private _console: Console,
  private _injector: Injector,
  private _exceptionHandler: ErrorHandler,
  private _componentFactoryResolver: ComponentFactoryResolver,
  private _initStatus: ApplicationInitStatus) {

  this._enforceNoNewChanges = isDevMode();

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

  const isCurrentlyStable =
    new Observable<boolean>((observer: Observer<boolean>) => {
    });

  const isStable = new Observable<boolean>((observer: Observer<boolean>)=>{
    // Create the subscription to onStable outside the Angular Zone so that
    // the callback is run outside the Angular Zone.
    let stableSub: Subscription;
    this._zone.runOutsideAngular(()
    });

    const unstableSub: Subscription = this._zone.onUnstable.subscribe();
  });

  return () => {

```

```

        stableSub.unsubscribe();
        unstableSub.unsubscribe();
    };
});

(this as {isStable: Observable<boolean>}).isStable =
    merge(isCurrentlyStable, share.call(isStable));

```

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>,
rootSelectorOrNode?: string|any):
    ComponentRef<C> {

    let componentFactory: ComponentFactory<C>;
    if (componentOrFactory instanceof ComponentFactory) {
        componentFactory = componentOrFactory;
    } else {
        componentFactory = this._componentFactoryResolver
            .resolveComponentFactory(componentOrFactory) !;
    }
    this.componentTypes.push(componentFactory.componentType);

    // Create a factory associated with the current module if
    // it's not bound to some other
    const ngModule = componentFactory instanceof
        ComponentFactoryBoundToModule ? Null :
        this._injector.get(NgModuleRef);
    const selectorOrNode = rootSelectorOrNode || componentFactory.selector;
    const compRef = componentFactory.create(Injector.NULL, [],
        selectorOrNode, ngModule);

    ..
    this._loadComponent(compRef);
    ..
    return compRef;
}

```

`_loadComponent()` is defined as:

```

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

```

Attached views are those that can be attached to a view container and are subject to dirty checking. Such views can be attached and detached, and an array of attached views is recorded.

```

private _views: InternalViewRef[] = [];
attachView(viewRef: ViewRef): void {
    const view = (viewRef as InternalViewRef);
    this._views.push(view);
}

```

```

    view.attachToAppRef(this);
  }
  detachView(viewRef: ViewRef): void {
    const view = (viewRef as InternalViewRef);
    remove(this._views, view);
    view.detachFromAppRef();
  }
  get viewCount() { return this._views.length; }

```

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/DependencyInjection Feature (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 {StaticProvider, ValueProvider, ExistingProvider, FactoryProvider,
  Provider, TypeProvider, ClassProvider} from './di/provider';
export {ResolvedReflectiveFactory, ResolvedReflectiveProvider}
  from './di/reflective_provider';
export {ReflectiveKey} from './di/reflective_key';
export {InjectionToken} from './di/injection_token';

```

The core/src/di directory contains these files:

- forward_ref.ts
- injection_token.ts
- injector.ts
- metadata.ts
- opaque_token.ts
- provider.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 Self: SelfDecorator = makeParamDecorator('Self');
export const SkipSelf: SkipSelfDecorator = makeParamDecorator('SkipSelf');
export const Inject: InjectDecorator = makeParamDecorator('Inject', (token:
any) => ({token}));
export const Optional: OptionalDecorator = makeParamDecorator('Optional');
export const Injectable: InjectableDecorator = makeDecorator('Injectable');
export const Host: HostDecorator = makeParamDecorator('Host');

```

Forward refs are placeholders used to facilitate 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();
  abstract get<T>(token: Type<T>|InjectionToken<T>, notFoundValue?: T): T;
  abstract get(token: any, notFoundValue?: any): any;
  static create(providers: StaticProvider[], parent?: Injector): Injector {

```

```

        return new StaticInjector(providers, parent);
    }
}

```

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.

`provider.ts` defines a number of `Provider` classes and then uses them to define the `Provider` type:

```

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

```

Core/Metadata Feature (core/src/metadata)

Think of metadata as little nuggets of information we would like to attach to other things. The [<ANGULAR-MASTER>/core/src/metadata.ts](#) file exports a variety of types from the `core/src/metadata` sub-directory:

```

export {ANALYZE_FOR_ENTRY_COMPONENTS, Attribute, ContentChild,
    ContentChildDecorator, ContentChildren, ContentChildrenDecorator,
    Query, ViewChild, ViewChildDecorator, ViewChildren,
    ViewChildrenDecorator} from './metadata/di';
export {Component, ComponentDecorator, Directive, DirectiveDecorator,
    HostBinding, HostListener, Input, Output, Pipe}
    from './metadata/directives';
export {AfterContentChecked, AfterContentInit, AfterViewChecked,
    AfterViewInit, DoCheck, OnChanges, OnDestroy, OnInit}
    from './metadata/lifecycle_hooks';
export {CUSTOM_ELEMENTS_SCHEMA, ModuleWithProviders, NO_ERRORS_SCHEMA,
    NgModule, SchemaMetadata} from './metadata/ng_module';
export {ViewEncapsulation} from './metadata/view';

```

The source files in the [<ANGULAR-MASTER>/packages/core/src/metadata](#) sub-directory are:

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

The `di.ts` file defines a range of interfaces for decorators – these include `AttributeDecorator`, `ContentChildrenDecorator`, `ContentChildDecorator`, `ViewChildrenDecorator`, `ViewChildDecorator`. It also defines and exports a number of consts that are implementations of those interfaces, created using `makePropDecorator`.

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

```

export enum ViewEncapsulation {
    Emulated=0,
    Native=1,
}

```



```

    None=2
  }

```

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 {@link Component#styles styles} or
 * {@link Component#styleUrls styleUrls}, and adding the new
 * Host Element attribute to all selectors.
 *
 * This is the default option.
 */

```

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

- DirectiveDecorator
- Directive
- ComponentDecorator
- Component
- PipeDecorator
- Pipe
- InputDecorator
- Input
- OutputDecorator
- Output
- HostBindingDecorator
- HostBinding
- HostListenerDecorator
- HostListener

The `lifecycle_hooks.ts` file defines a number of interfaces used for lifecycle hooks:

```

export interface SimpleChanges { [propName: string]: SimpleChange; }
export interface OnChanges { ngOnChanges(changes: SimpleChanges): void; }
export interface OnInit { ngOnInit(): void; }
export interface DoCheck { ngDoCheck(): void; }
export interface OnDestroy { ngOnDestroy(): void; }
export interface AfterContentInit { ngAfterContentInit(): void; }
export interface AfterContentChecked { ngAfterContentChecked(): void; }
export interface AfterViewInit { ngAfterViewInit(): void; }
export interface AfterViewChecked { ngAfterViewChecked(): void; }

```

These define the method signatures for handlers that application component interest in the lifecycle hooks must implement.

The `ng_module.ts` file contains constructs used to define Angular modules. An Angular application is built from a set of these and `NgModule` is used to tie them together:

```

export interface NgModule {
  providers?: Provider[];
  declarations?: Array<Type<any>|any[]>;
  imports?: Array<Type<any>|ModuleWithProviders|any[]>;
  exports?: Array<Type<any>|any[]>;
  entryComponents?: Array<Type<any>|any[]>;
}

```

```
bootstrap?: Array<Type<any>|any[]>;
schemas?: Array<SchemaMetadata|any[]>;
id?: string;
}
```

An `NgModuleDecorator` is how `NgModule` is attached to a type:

```
export interface NgModuleDecorator {
  (obj?: NgModule): TypeDecorator;
  new (obj?: NgModule): NgModule;
}
```

We see their usage when we explore the boilerplate code that Angular CLI generates for us when we use it to create a new Angular project. Look inside

- `<my-project>/src/app/app.module.ts`

and we will see:

```
@NgModule({
  declarations: [
    <your-components>
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [
    Title,
    <your-custom-providers>
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

and when we look at:

- `<my-project>/src/main.ts`

we see:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
..
platformBrowserDynamic().bootstrapModule(AppModule)
    .catch(err => console.log(err));
```

Core/Profile Feature (core/src/profile)

The profile directory has these files:

- `profile.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>

Its github project is here:

- <https://github.com/google/tracing-framework>

wtf_impl.ts define the following interfaces:

```
export interface WtfScopeFn { (arg0?: any, arg1?: any): any; }
interface WTF { trace: Trace; }
interface Trace {
  events: Events;
  leaveScope(scope: Scope, returnValue: any): any;
  beginTimeRange(rangeType: string, action: string): Range;
  endTimeRange(range: Range): any;
}
export interface Range {}
interface Events {
  createScope(signature: string, flags: any): Scope;
}
export interface Scope { (...args: any[]): any; }
```

It maintains two variables:

```
let trace: Trace;
let 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): void;
export function leave<T>(scope: Scope, returnValue?: T): T;
export function leave<T>(scope: Scope, returnValue?: any): any {
  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 {
  const 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 {WtfScopeFn} from './wtf_impl';
export const wtfEnabled = detectWTF();
export const wtfCreateScope: (signature: string, flags?: any) => WtfScopeFn =
  wtfEnabled ? createScope : (signature: string, flags?: any) => noopScope;
export const wtfLeave: <T>(scope: any, returnValue?: T) => T =
  wtfEnabled ? leave : (s: any, r?: any) => r;
export const wtfStartTimeRange: (rangeType: string, action: string) => any =
  wtfEnabled ? startTimeRange : (rangeType: string, action: string) =>
  null;
export const wtfEndTimeRange: (range: any) => void = wtfEnabled ?
  endTimeRange : (r: any) => null;

```

Core Reflection Feature (core/src/reflection)

The reflection directory has these files:

- platform_reflection_capabilities.ts
- reflection_capabilities.ts
- reflection.ts
- reflector.ts
- types.ts

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 these lines:

```

export {Reflector} from './reflector';
export const reflector = new Reflector(new ReflectionCapabilities());

```

The platform_reflection_capabilities.ts file defines this interface:

```

export interface PlatformReflectionCapabilities {
  isReflectionEnabled(): boolean;
  factory(type: Type<any>): Function;
  hasLifecycleHook(type: any, lcProperty: string): boolean;
  parameters(type: Type<any>): any[][];
  annotations(type: Type<any>): any[];
  propMetadata(typeOrFunc: Type<any>): {[key: string]: any[]};
  getter(name: string): GetterFn;
  setter(name: string): SetterFn;
  method(name: string): MethodFn;
  importUri(type: Type<any>): string;
  resourceUri(type: Type<any>): string;
  resolveIdentifier(name: string, moduleId: string,
    members: string[], runtime: any): any;
  resolveEnum(enumIdentifier: any, name: string): any;
}

```

The reflection_capabilities.ts provides an implementation of that interface called ReflectionCapabilities. We see the main part of reflection is in the reflector.ts file, which defines the Reflector class – its methods mostly forward calls to the supplied ReflectionCapabilities instance:

```

/**
 * Provides access to reflection data about symbols.
 * Used internally by Angular to power dependency injection and compilation.

```

```

*/
export class Reflector {
  constructor(public reflectionCapabilities:
    PlatformReflectionCapabilities) {}
  updateCapabilities(caps: PlatformReflectionCapabilities)
    { this.reflectionCapabilities = caps; }
  factory(type: Type<any>): Function
    { return this.reflectionCapabilities.factory(type); }
  parameters(typeOrFunc: Type<any>): any[][]
    { return this.reflectionCapabilities.parameters(typeOrFunc); }
  annotations(typeOrFunc: Type<any>): any[]
    { return this.reflectionCapabilities.annotations(typeOrFunc); }
  propMetadata(typeOrFunc: Type<any>): {[key: string]: any[]}
    { return this.reflectionCapabilities.propMetadata(typeOrFunc); }
  hasLifecycleHook(type: any, lcProperty: string): boolean
    { return this.reflectionCapabilities.hasLifecycleHook(type, lcProperty); }
  getter(name: string): GetterFn
    { return this.reflectionCapabilities.getter(name); }
  setter(name: string): SetterFn
    { return this.reflectionCapabilities.setter(name); }
  method(name: string): MethodFn
    { return this.reflectionCapabilities.method(name); }
  importUri(type: any): string
    { return this.reflectionCapabilities.importUri(type); }
  resourceUri(type: any): string
    { return this.reflectionCapabilities.resourceUri(type); }
  resolveIdentifier(name: string, moduleUrl: string,
    members: string[], runtime: any): any {
    return this.reflectionCapabilities.resolveIdentifier(
      name, moduleUrl, members, runtime);
  }
  resolveEnum(identifier: any, name: string): any {
    return this.reflectionCapabilities.resolveEnum(identifier, name);
  }
}

```

Core Render Feature (core/src/render)

Note: There is a new view compilation and rendering architecture (called Render3, better known by its code name, Ivy) under intensive development by the Angular team. A good place to keep an eye on progress and how it is evolving is:

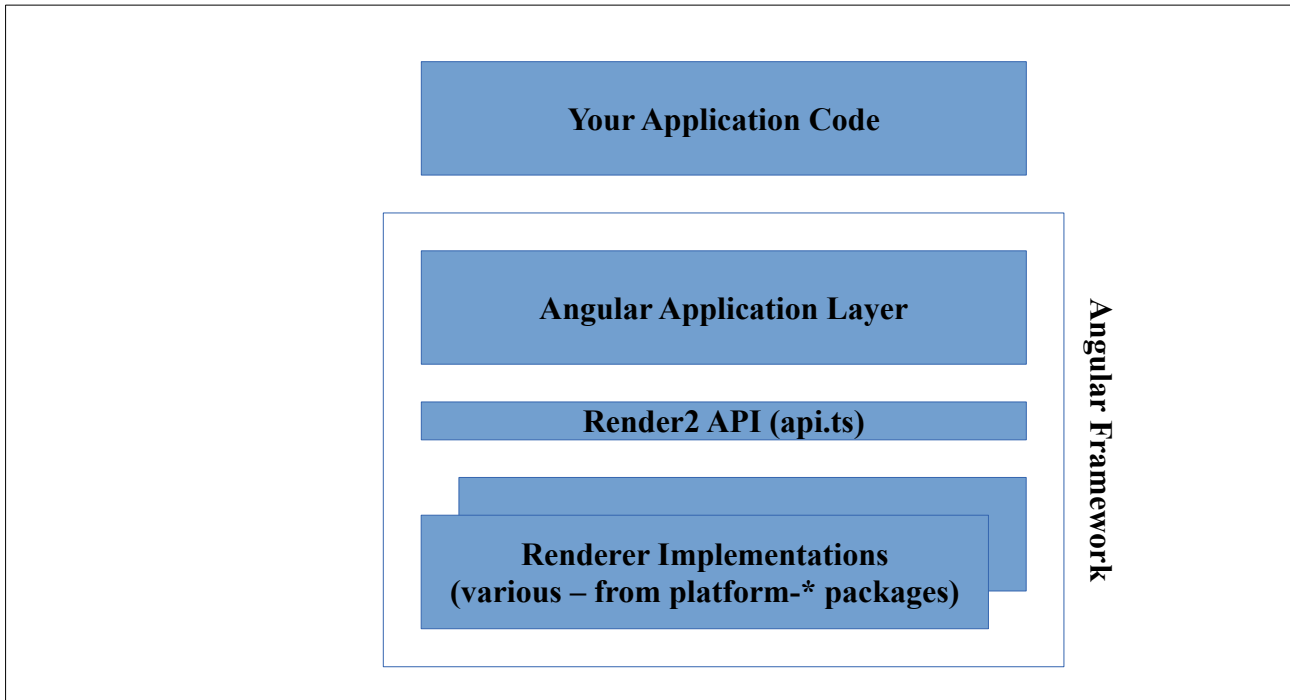
<https://github.com/angular/angular/tree/master/packages/core/src/render3>

What we describe below is Render2, which is the current stable view engine. Please see the appendix for detailed coverage of Render3.

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 these abstract classes - `RendererType2`, `Renderer2`, `RendererFactory2` and `RendererStyleFlags2`. Implementation of this API is not part of Core. Instead, the various Platform-X packages need to provide the actual implementations for different scenarios.

Scenarios with diverse rendering requirements include:

- UI web apps in regular browser
- web worker 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 web worker 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-MASTER>/packages/compiler/src/schema](https://github.com/angular/angular/blob/master/packages/compiler/src/schema)) and we will examine it further when looking at the template compiler.

Now we will move on to looking at the renderer API. This API is exported from the

- [<ANGULAR-MASTER>/packages/core/src/render.ts](https://github.com/angular/angular/blob/master/packages/core/src/render.ts)

and it contains this export:

```
// Public API for render
```

```
export {RenderComponentType, Renderer, Renderer2, RendererFactory2,
  RendererStyleFlags2, RendererType2, RootRenderer} from './render/api';
```

We note a number of the exports have "2" in their name and when we examine the actual definitions we see those that do not are deprecated so will not be covering those here. If looking at older documentation you may encounter them, but specifically for rendering, for up to date code you should be looking at rendering classes with 2 in the name. Also note '2' for rendering is not tied to Angular version 2 (actually that version of Angular used the older version of the rendering APIs).

The core/src/render directory has just one file:

- [<ANGULAR-MASTER>/packages/core/src/render/api.ts](#)

and this contains the rendering definitions we will see used elsewhere by platform-implementations.

RendererType2 is used to identify component types for which rendering is needed, and is defined as:

```
/**
 * Used by `RendererFactory2` to associate custom rendering data and styles
 * with a rendering implementation.
 */
export interface RendererType2 {
  id: string;
  encapsulation: ViewEncapsulation;
  styles: (string|any[])[];
  data: {[kind: string]: any};
}
```

The `RendererFactory2` class is used to register a provider with dependency injection and is defined as:

```
/**
 * Creates and initializes a custom renderer that implements
 * the `Renderer2` base class.
 */
export abstract class RendererFactory2 {
  abstract createRenderer(
    hostElement: any, type: RendererType2|null): Renderer2;
  abstract begin?(): void;
  abstract end?(): void;
  abstract whenRenderingDone?(): Promise<any>;
}
```

Essentially, its main method, `createRenderer()`, is used to answer this - "for a given host element, please give me back a renderer that I can use". This is key to wiring up flexible rendering of components via dependency injection.

An example usage is in:

- [<ANGULAR-MASTER>/packages/platform-browser/src/browser.ts](#)

where `browserModule` is defined as:

```
export const BROWSER_MODULE_PROVIDERS: StaticProvider[] = [
  ...
  {
```

```

    provide: DomRendererFactory2,
    useClass: DomRendererFactory2,
    deps: [EventManager, DomSharedStylesHost, APP_ID]
  },
  {provide: RendererFactory2, useExisting: DomRendererFactory2},

```

Another example usage is in:

- [<ANGULAR-MASTER>/packages/platform-webworker/src/worker_app.ts](#)

where WorkerAppModule is defined as:

```

@NgModule({
  providers: [
    ..
    WebWorkerRendererFactory2,
    {provide: RendererFactory2, useExisting: WebWorkerRendererFactory2},
    RenderStore,
    ..
  ],
  ..
})
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 `RendererFactory2` gets into dependency injection system, then of course the next question is, how does it get out?

The file

- [<ANGULAR-MASTER>/packages/core/src/view/services.ts](#)

has this:

```

function createProdRootView(
  elInjector: Injector, projectableNodes: any[][],
  rootSelectorOrNode: string | any,
  def: ViewDefinition, ngModule: NgModuleRef<any>, context?: any): ViewData {
  const rendererFactory: RendererFactory2 =
    ngModule.injector.get(RendererFactory2);

  return createRootView(
    createRootData(elInjector, ngModule, rendererFactory,
      projectableNodes, rootSelectorOrNode),
    def, context);
}

```

which is called from:

```

function createProdServices() {
  return {
    setCurrentNode: () => {},
    createRootView: createProdRootView, ...
  }
}

```

which in turn is called from:

```

export function initServicesIfNeeded() {
  if (initialized) {
    return;
  }
}

```



```

    }
    initialized = true;
    const services = isDevMode() ? createDebugServices() :
    createProdServices();

```

which is finally called from inside:

- [<ANGULAR-MASTER>/packages/core/src/view/entrypoint.ts](#)

in `NgModuleFactory_.create()` method:

```

class NgModuleFactory_ extends NgModuleFactory<any> {
  ..
  create(parentInjector: Injector|null): NgModuleRef<any> {
    initServicesIfNeeded();
    const def = cloneNgModuleDefinition(
      resolveDefinition(this._ngModuleDefFactory));
    return Services.createNgModuleRef(
      this.moduleType, parentInjector || Injector.NULL,
      this._bootstrapComponents, def);
  }
}

```

Now we'll return to:

- [<ANGULAR-MASTER>/packages/core/src/render/api.ts](#)

and move on to the principal class in the Renderer API, `Renderer2`, which is abstract and declares the following methods:

get data	insertBefore	addClass
destroy	removeChild	removeClass
createElement	selectRootElement	setStyle
createComment	parentNode	removeStyle
createText	nextSibling	setProperty
destroyNode	setAttribute	setValue
appendChild	removeAttribute	listen

`Renderer2` in full is as follows:

```

export abstract class Renderer2 {
  abstract get data(): {[key: string]: any};
  abstract destroy(): void;
  abstract createElement(name: string, namespace?: string|null): any;
  abstract createComment(value: string): any;
  abstract createText(value: string): any;
  abstract destroyNode !: ((node: any) => void) | null;
  abstract appendChild(parent: any, newChild: any): void;
  abstract insertBefore(parent: any, newChild: any, refChild: any): void;
  abstract removeChild(parent: any, oldChild: any,
    isHostElement?: boolean): void;
  abstract selectRootElement(
    selectorOrNode: string|any, preserveContent?: boolean): any;
  abstract parentNode(node: any): any;
  abstract nextSibling(node: any): any;
  abstract setAttribute(el: any, name: string, value: string,
    namespace?: string|null): void;
  abstract removeAttribute(el: any, name: string,
    namespace?: string|null): void;
  abstract addClass(el: any, name: string): void;

```

```

    abstract removeClass(el: any, name: string): void;
    abstract setStyle(el: any, style: string, value: any, flags?:
RendererStyleFlags2): void;
    abstract removeStyle(el: any,
        style: string, flags?: RendererStyleFlags2): void;
    abstract setProperty(el: any, name: string, value: any): void;
    abstract setValue(node: any, value: string): void;
    abstract listen(
        target: 'window'|'document'|'body'|any, eventName: string,
        callback: (event: any) => boolean | void): () => void;
    static __NG_ELEMENT_ID__:
        () => Renderer2 = () => SWITCH_RENDERER2_FACTORY();
}

```

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 engine implementations – from the Angular team and third parties.

Finally is `RendererStyleFlags2` defined as:

```

export enum RendererStyleFlags2 {
    Important = 1 << 0,
    DashCase = 1 << 1
}

```

It is used to supply a flag parameter to two of `Renderer2`'s methods:

```

abstract setStyle(el: any, style: string, value: any,
    flags?: RendererStyleFlags2): void;
abstract removeStyle(el: any, style: string,
    flags?: RendererStyleFlags2): void;

```

core/src/debug

This directory contains this file:

- `debug_node.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|null;

    constructor(nativeNode: any, parent: DebugNode|null,
        private _debugContext: DebugContext) {
        this.nativeNode = nativeNode;
        if (parent && parent instanceof DebugElement) {
            parent.addChild(this);
        } else {

```

```

    this.parent = null;
  }
  this.listeners = [];
}
get injector(): Injector { return this._debugContext.injector; }
get componentInstance(): any { return this._debugContext.component; }
get context(): any { return this._debugContext.context; }
get references(): {[key: string]: any} {
  return this._debugContext.references; }
get providerTokens(): any[] { return this._debugContext.providerTokens; }
}

```

The debug node is attached as a child to the parent `DebugNode`. The `nativeNode` to which this `DebugNode` refers to is recorded. A private field, `_debugContext` records and supplies additional debugging context. It is of type `DebugContext`, defined in `view/types.ts` as:

```

export abstract class DebugContext {
  abstract get view(): ViewData;
  abstract get nodeIndex(): number|null;
  abstract get injector(): Injector;
  abstract get component(): any;
  abstract get providerTokens(): any[];
  abstract get references(): {[key: string]: any};
  abstract get context(): any;
  abstract get componentRenderElement(): any;
  abstract get renderNode(): any;
  abstract logError(console: Console, ...values: any[]): void;
}

```

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 | null};
  classes: {[key: string]: boolean};
  styles: {[key: string]: string | null};
  childNodes: DebugNode[];
  nativeElement: any;

  constructor(nativeNode: any, parent: any, _debugContext: DebugContext) {
    super(nativeNode, parent, _debugContext);
    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 (child) {
    this.childNodes.push(child);
    child.parent = this;
  }
}

```

```
removeChild(child: DebugNode) {
  const 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.
const _nativeNodeToDebugNode = new Map<any, DebugNode>();
```

This is used to add a node:

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

Core ChangeDetection Feature (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=0,
  Default=1,
}
export enum ChangeDetectorStatus {
  CheckOnce,
  Checked,
  CheckAlways,
  Detached,
  Errored,
  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);
```

The `differs` sub-directory provides the actual implementation for the change detection algorithms, in these files:

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

Core/Zone Feature (core/src/zone)

Source Tree

This directory contains this file:

- `ng_zone.ts`

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

The `ng_zone.ts` file in Angular Core defines the `NgZone` class, which provides an Angular wrapper for zones. It is defined as:

```
export class NgZone {
  readonly hasPendingMicrotasks: boolean = false;
  readonly hasPendingMacrotasks: boolean = false;
  readonly isStable: boolean = true;
  readonly onUnstable: EventEmitter<any> = new EventEmitter(false);
  readonly onMicrotaskEmpty: EventEmitter<any> = new EventEmitter(false);
  readonly onStable: EventEmitter<any> = new EventEmitter(false);
  readonly onError: EventEmitter<any> = new EventEmitter(false);
  constructor({enableLongStackTrace = false}) {...}
  static isInAngularZone(): boolean {...}
  static assertInAngularZone(): void {...}
  static assertNotInAngularZone(): void {...}
  run<T>(fn: (...args: any[]) => T,
```

```

    applyThis?: any, applyArgs?: any[]): T {...}
  runTask<T>(fn: (...args: any[]) => T,
    applyThis?: any, applyArgs?: any[], name?: string): T {...}
  runGuarded<T>(fn: (...args: any[]) => T,
    applyThis?: any, applyArgs?: any[]): T {...}
  runOutsideAngular<T>(fn: (...args: any[]) => T): T {...}
}

```

Its constructor implementation is worth studying. First note Angular requires Zone.js and throws an error if it is not found. If `wtfZoneSpec` or `longStackTraceZoneSpec` zones are required, it forks as needed:

```

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

  Zone.assertZonePatched();
  const self = this as any as NgZonePrivate;
  self._nesting = 0;

  self._outer = self._inner = Zone.current;
  if ((Zone as any)['wtfZoneSpec']) {
    self._inner = self._inner.fork((Zone as any)['wtfZoneSpec']);
  }
  if (enableLongStackTrace && (Zone as any)['longStackTraceZoneSpec']) {
    self._inner = self._inner.fork((Zone as any)
      ['longStackTraceZoneSpec']);
  }
  forkInnerZoneWithAngularBehavior(self);
}

```

Core/Testability Feature (core/src/testability)

Testability is the provision of testing hooks that can be used by test-focused tooling, such as Angular's Protractor.

Core/Testability Public API

The public API for this feature is exported by [core/src/core.ts](#):

```

export {GetTestability, Testability, TestabilityRegistry,
  setTestabilityGetter} from './testability/testability';

```

It exports consists of an interface – `GetTestability`, a function – `setTestabilityGetter`, and two injectable classes, `TestabilityRegistry` and `Testability`.

Core/Testability Usage

We see use of `TestabilityRegistry` in [core/src/platform_core_providers.ts](#) by `_CORE_PLATFORM_PROVIDERS`:

```

const _CORE_PLATFORM_PROVIDERS: StaticProvider[] = [
  ..
  {provide: TestabilityRegistry, deps: []},

```

The bootstrap method in `ApplicationRef` looks for configured `Testability` elements via dependency injection **1** and if found **2**, then registers them **3** with the

TestabilityRegistry (which itself is also accessed via dependency injection):

```

@Injectable() constructor(
  export class ApplicationRef {
    bootstrap<C>(..):ComponentRef<C> {
      ..
      const compRef = componentFactory.create(..);
      ..
1     const testability = compRef.injector.get(Testability, null);
2     if (testability) {
3       compRef.injector.get(TestabilityRegistry)
        .registerApplication(compRef.location.nativeElement, testability);
      }
    }
  }
}

```

Note there is a class BrowserGetTestability in:

- [<ANGULAR-MASTER>/packages/platform-browser/src/browser/testability.ts](https://github.com/angular/angular/blob/master/packages/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/Testability Implementation

The implementation of the testability feature is in the testability sub-directory, which contains two file:

- testability.externs.js
- testability.ts

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

GetTestability is simply defined as:

```

/**
 * Adapter interface for retrieving the `Testability` service associated
 * for a particular context.
 *
 * @experimental Testability apis are primarily intended to be used by
 * e2e test tool vendors like the Protractor team.
 */
export interface GetTestability {
  addToWindow(registry: TestabilityRegistry): void;
  findTestabilityInTree(registry: TestabilityRegistry, elem: any,
    findInAncestors: boolean): Testability|null;
}

```

setTestabilityGetter is used to set the getter:

```

/**
 * Set the GetTestability implementation used by
 * the Angular testing framework.
 */
export function setTestabilityGetter(getter: GetTestability): void {
  _testabilityGetter = getter;
}

```

The `TestabilityRegistry` class is “A global registry of Testability instances for specific elements”. `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:

```
@Injectable()
export class TestabilityRegistry {
  _applications = new Map<any, Testability>();
  constructor() { _testabilityGetter.addToWindow(this); }
  registerApplication(token: any, testability: Testability) {
    this._applications.set(token, testability);
  }
  unregisterApplication(token: any) { this._applications.delete(token); }
  unregisterAllApplications() { this._applications.clear(); }
  getTestability(elem: any): Testability|null {
    return this._applications.get(elem) || null; }
  getAllTestabilities(): Testability[] {
    return Array.from(this._applications.values()); }
  getAllRootElement(): any[] { return Array.from(this._applications.keys()); }
}
  findTestabilityInTree(elem: Node, findInAncestors: boolean = true):
Testability|null {.. }
}
```

The `Testability` class is structured as follows:

```
/**
 * The Testability service provides testing hooks that can be accessed from
 * the browser and by services such as Protractor. Each bootstrapped Angular
 * application on the page will have an instance of Testability.
 */
@Injectable()
export class Testability implements PublicTestability {
  _pendingCount: number = 0;
  _isZoneStable: boolean = true;
  /**
   * Whether any work was done since the last 'whenStable' callback. This is
   * useful to detect if this could have potentially destabilized another
   * component while it is stabilizing.
   */
  _didWork: boolean = false;
  _callbacks: Function[] = [];
  constructor(private _ngZone: NgZone) { this._watchAngularEvents(); }
  _watchAngularEvents(): void { }
  increasePendingRequestCount(): number { }
  decreasePendingRequestCount(): number { }
  isStable(): boolean { }
  whenStable(callback: Function): void { }
  getPendingRequestCount(): number { return this._pendingCount; }
  findProviders(using: any, provider: string, exactMatch: boolean): any[] { }
}
```

Core/Linker Feature (core/src/linker)

The Core/Linker feature is used to define an API for the template compiler and is instrumental in how compiled components work together.

When developers new to Angular hear about the linker, their mind immediately races

to the conclusion that it is something like the C linker or linkers for other languages, but that is far from the case with Angular's linker. Remember Angular is dealing with template syntax (a quasi-HTML representation of a page which includes embedded TypeScript logic) and this needs to work with component code written in TypeScript. So conceptually the angular linker is bringing different parts of your application together (so using the term "linker" is technically correct), but in practice how it works is quite different to what happens with an ANSI C linker.

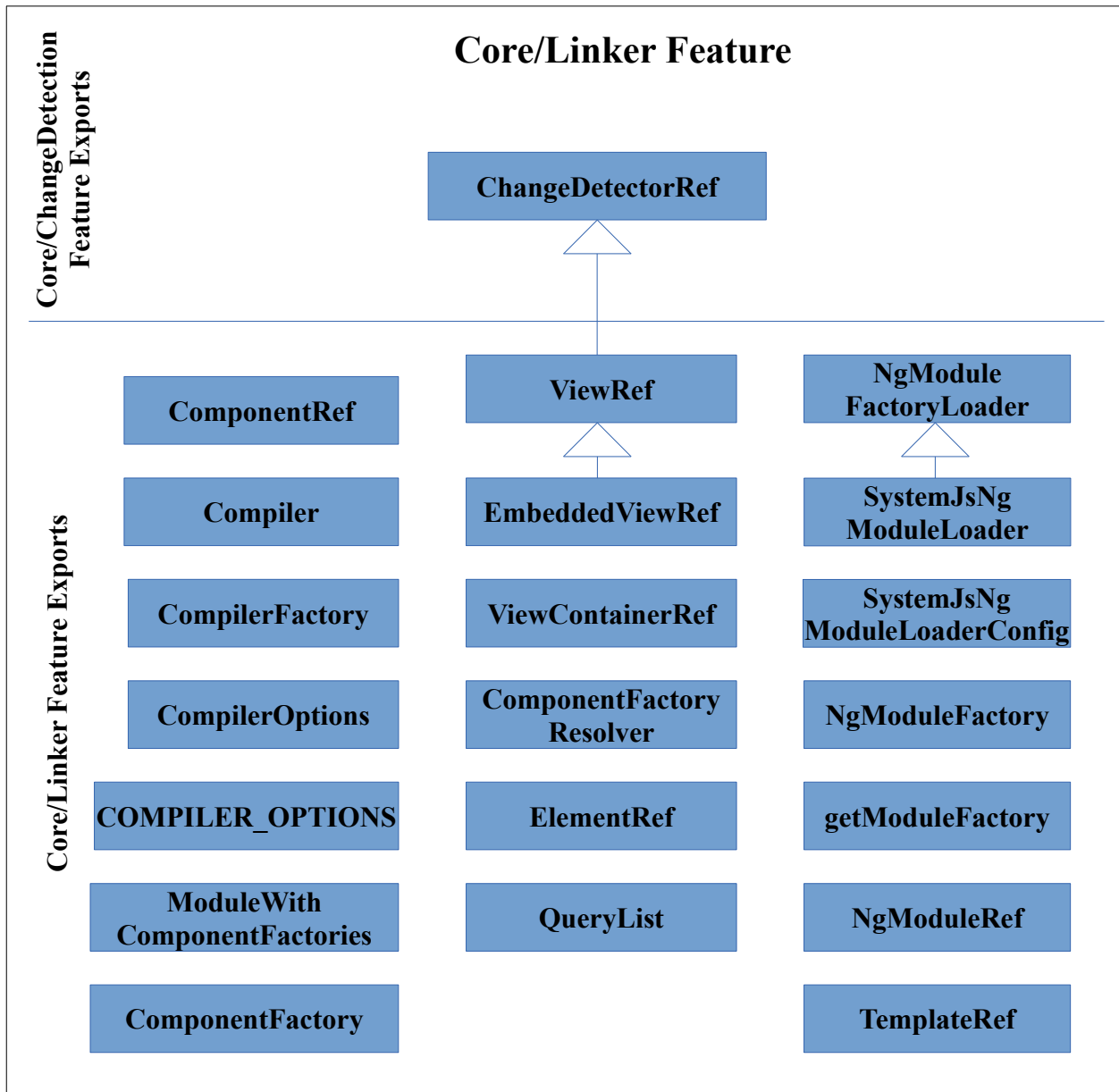
Core/Linker Public API

The public API for this feature is exported by the [<ANGULAR-MASTER>/packages/core/src/core.ts](#) file, which has this line:

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

The [<ANGULAR-MASTER>/packages/core/src/linker.ts](#) file lists the exports:

```
// Public API for compiler
export {COMPILER_OPTIONS, Compiler, CompilerFactory, CompilerOptions,
  ModuleWithComponentFactories} from './linker/compiler';
export {ComponentFactory, ComponentRef} from './linker/component_factory';
export {ComponentFactoryResolver} from './linker/component_factory_resolver';
export {ElementRef} from './linker/element_ref';
export {NgModuleFactory, NgModuleRef} from './linker/ng_module_factory';
export {NgModuleFactoryLoader, getModuleFactory}
  from './linker/ng_module_factory_loader';
export {QueryList} from './linker/query_list';
export {SystemJsNgModuleLoader, SystemJsNgModuleLoaderConfig}
  from './linker/system_js_ng_module_factory_loader';
export {TemplateRef} from './linker/template_ref';
export {ViewContainerRef} from './linker/view_container_ref';
export {EmbeddedViewRef, ViewRef} from './linker/view_ref';
```



As we see, there is one enum (`COMPILER_OPTIONS`), one method (`getModuleFactory`), one type (`CompilerOptions`) and the other public exports are classes. We note there is not much hierarchy in the public API layout.

Core/Linker Usage

core/Linker is really the base API for compilation and linking, but we will see these types being used elsewhere (in particular, in the Angular Compiler package). The source files in the Core/Linker feature are only a few hundred lines of code in total. The heavy lifting of compilation is done in the Compiler package, which is much, much larger. For example, a single one of its file, `view_compiler.ts`, is over a thousand lines long. We also see Core/Linker being used by the Compiler-CLI package, which is a command-line interface to the Compiler package.

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: Provider[] = [
  ..
  {provide: NgModuleFactoryLoader, useClass: SystemJsNgModuleLoader} ];
```

Core/Linker Implementation

The source files in [core/src/linker](#) implement the Core/Linker feature:

- `compiler.ts`
- `component_factory_resolver.ts`
- `component_factory.ts`
- `element_ref.ts`
- `ng_module_factory_loader.ts`
- `ng_module_factory.ts`
- `query_list.ts`
- `system_js_ng_module_factory_loader.ts`
- `template_ref.ts`
- `view_container_ref.ts`
- `view_ref.ts`

A number of these files introduce types whose names end in `Ref`:

- `ElementRef`
- `TemplateRef`
- `ViewRef`
- `ViewContainerRef`
- `ComponentRef`

If you are used to programming in a language such as C# where the phrase reference type has a specific meaning, note that in Angular this is not a language concept, and more a naming convention. These references are merely collections of methods and properties used to interact with the referenced construct.

The elements in the DOM are exposed to Angular apps as `ElementRefs`. In general, it is recommended not to work directly with `ElementRefs`, as a certain level of abstraction is very useful to enable Angular apps work on different rendering targets.

As Angular application developers, the template syntax files we create are processed by the Angular template compiler to produce a number of template refs. If a template representation has a simple set of elements (`<p>hello world</p>`) then a single `TemplateRef` will be created. If directives such as `NgFor` are used then the outer content will be in one `TemplateRef` and what is inside the `NgFor` will be in another. A `TemplateRef` is instantiated one or more times to make a `ViewRef`. There is not necessarily a one-to-one mapping between `TemplateRefs` and `ViewRefs`. Depending on the number of iterations to be applied to an `NgFor`, there will be that number of instances of `ViewRefs`, all based on the same `TemplateRef`.

A view is a hierarchy and child views can be inserted using `ViewContainerRef` (literally a container for a child view). A view is the unit of hierarchy that Angular exposes to app developers to dynamically modify what is displayed to users. So think

in terms of adding and removing embedded views, not adding and removing individual HTML elements.

A `ComponentRef` contains a `HostView` which in turn can contain a number of embedded views.

Now we will move on to looking at the source, starting with looking at [core/src/linker/compiler.ts](#). `CompilerOptions` provides a list of configuration options for a compiler (all are marked as optional):

```
export type CompilerOptions = {
  useJit?: boolean,
  defaultEncapsulation?: ViewEncapsulation,
  providers?: StaticProvider[],
  missingTranslation?: MissingTranslationStrategy,
  enableLegacyTemplate?: boolean,
  preserveWhitespaces?: boolean,
};
```

`COMPILER_OPTIONS` is an exported const:

```
// Token to provide CompilerOptions in the platform injector.
export const COMPILER_OPTIONS =
  new InjectionToken<CompilerOptions[]>('compilerOptions');
```

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

```
export abstract class CompilerFactory {
  abstract createCompiler(options?: CompilerOptions[]): Compiler;
}
```

`ModuleWithComponentFactories` combines a `NgModuleFactory` and a `ComponentFactory`:

```
// Combination of NgModuleFactory and ComponentFactory.
export class ModuleWithComponentFactories<T> {
  constructor(
    public ngModuleFactory: NgModuleFactory<T>,
    public componentFactories: ComponentFactory<any>[] {}
  }
}
```

The `Compiler` class is used to perform template compilation:

```
@Injectable()
export class Compiler {
  compileModuleSync<T>(moduleType: Type<T>): NgModuleFactory<T> { .. }
  compileModuleAsync<T>(moduleType: Type<T>):
    Promise<NgModuleFactory<T>> { .. }
  compileModuleAndAllComponentsSync<T>(moduleType: Type<T>):
    ModuleWithComponentFactories<T> { .. }
  compileModuleAndAllComponentsAsync<T>(moduleType: Type<T>):
    Promise<ModuleWithComponentFactories<T>> { .. }
  clearCache(): void {}
  clearCacheFor(type: Type<any>) {}
}
```

The two methods `clearCache` and `clearCacheFor` are both empty, the other methods just throw an error. So to use a compiler, an actual implementation is needed and

that is where the separate Compiler package comes in. The `Compiler` class defined here is the base class for template compilers, and it is these derived classes where the actual template compilation occurs.

The four compile generic methods either synchronously or asynchronously compile an individual component, or an entire `NgModule`. All take a `Type<T>` as a parameter. The async versions returns the promise of one or more factories, whereas the sync versions return the actual factories.

The `query_list.ts` file defines the generic `QueryList<T>` class, which provides controlled access to a read-only array of items of type `T`.

The `ElementRef` class is used as a reference to the actual rendered element - what that is depends on the renderer.

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

In general, application developers are advised not to work directly with `ElementRefs`, as it makes their code renderer-specific, and may introduce security issues. We bring your attention to these lines in the source:

```
* @security Permitting direct access to the DOM can make your application more
* vulnerable to XSS attacks. Carefully review any use of `ElementRef` in your
* code. For more detail, see the [Security Guide](http://g.co/ng/security).
```

The `view_ref.ts` file defines the `ViewRef` and `EmbeddedViewRef` abstract classes which are exported by the package's [core/src/core.ts](#) file (via [core/src/linker.ts](#)), and an interface, `InternalViewRef`, that is used internally with the core package itself.

`ViewRef` is defined as:

```
export abstract class ViewRef extends ChangeDetectorRef {
  abstract destroy(): void;
  abstract get destroyed(): boolean;
  abstract onDestroy(callback: Function): any;
}
```

`EmbeddedViewRef` is defined as:

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

`InternalViewRef` is defined as:

```
export interface InternalViewRef extends ViewRef {
  detachFromAppRef(): void;
  attachToAppRef(appRef: ApplicationRef): void;
}
```

The `attachToAppRef/detachFromAppRef` methods are called from [ApplicationRef's](#) `attachView` and `detachView` methods:

```
attachView(viewRef: ViewRef): void {
```

```

    const view = (viewRef as InternalViewRef);
    this._views.push(view);
    view.attachToAppRef(this);
  }
  detachView(viewRef: ViewRef): void {
    const view = (viewRef as InternalViewRef);
    remove(this._views, view);
    view.detachFromAppRef();
  }

```

The `view_container_ref.ts` file defines the abstract `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). It defines two important create methods – `createEmbeddedView` and `createComponent`, which create the two variants of views supported. Finally, it has a few abstract helper methods – `clear`, `get`, `insert`, `indexOf`, `remove` and `detach` – which work on the views within the container.

```

export abstract class ViewContainerRef {
  abstract get element(): ElementRef;
  abstract get injector(): Injector;
  abstract get parentInjector(): Injector;
  abstract clear(): void;
  abstract get(index: number): ViewRef|null;
  abstract get length(): number;
  abstract createEmbeddedView<C>(templateRef: TemplateRef<C>,
    context?: C, index?: number): EmbeddedViewRef<C>;
  abstract createComponent<C>(
    componentFactory: ComponentFactory<C>,
    index?: number, injector?: Injector,
    projectableNodes?: any[][],
    ngModule?: NgModuleRef<any>): ComponentRef<C>;
  abstract insert(viewRef: ViewRef, index?: number): ViewRef;
  abstract move(viewRef: ViewRef, currentIndex: number): ViewRef;
  abstract indexOf(viewRef: ViewRef): number;
  abstract remove(index?: number): void;
  abstract detach(index?: number): ViewRef|null;
}

```

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.

The `component_factory.ts` file exports two abstract classes – `ComponentRef` and `ComponentFactory`:

```

export abstract class ComponentFactory<C> {}
  abstract get selector(): string;
  abstract get componentType(): Type<any>;
  abstract get ngContentSelectors(): string[];
  abstract get inputs(): {propName: string, templateName: string}[];
  abstract get outputs(): {propName: string, templateName: string}[];
  abstract create(
    injector: Injector, projectableNodes?: any[][], rootSelectorOrNode?:

```

The `ComponentRef` abstract class is defined as:

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

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

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

The `component_factory_resolver.ts` file defines the `ComponentFactoryResolver` abstract class, which is part of the public API:

`ComponentFactoryResolver` is defined as:

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

this file also defines `CodegenComponentFactoryResolver`, a concrete class that is exported via:

- [/packages/core/src/codegen_private_exports.ts](https://github.com/angular/angular/blob/master/packages/core/src/codegen_private_exports.ts)

as it is used elsewhere within the Angular ecosystem.

`CodegenComponentFactoryResolver` is defined as:

```
export class CodegenComponentFactoryResolver implements
ComponentFactoryResolver {
  1 private _factories = new Map<any, ComponentFactory<any>>();

  constructor(
    2 factories: ComponentFactory<any>[],
    private _parent: ComponentFactoryResolver,
    private _ngModule: NgModuleRef<any>) {
    3 for (let i = 0; i < factories.length; i++) {
      const factory = factories[i];
      this._factories.set(factory.componentType, factory);
    }
  }

  4 resolveComponentFactory<T>(
    component: {new (...args: any[]): T}): ComponentFactory<T> {
    5 let factory = this._factories.get(component);
    if (!factory && this._parent) {
    6 factory = this._parent.resolveComponentFactory(component);
    }
  }
}
```

```

    if (!factory) {
        throw noComponentFactoryError(component);
    }
    7 return new ComponentFactoryBoundToModule(factory, this._ngModule);
}
}

```

CodegenComponentFactoryResolver has a private map field, `_factories` **1**, and its constructor takes an array, `factories` **2**. Don't mix them up! The constructor iterates **3** 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()` **4** method, `CodegenComponentFactoryResolver` looks up the map for a matching factory, and if present **5** selects that as the factory and if not, calls the `resolveComponentFactory` method **6** of the parent, and passes the selected factory to a new instance of the `ComponentFactoryBoundToModule` helper class **7**.

The `ng_module_factory_loader.ts` file defines the `NgModuleFactoryLoader` abstract class as:

```

export abstract class NgModuleFactoryLoader {
    abstract load(path: string): Promise<NgModuleFactory<any>>;
}

```

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

The `system_js_ng_module_factory_loader.ts` file defines the `SystemJsNgModuleLoader` class, which loads `NgModule` factories using `SystemJS`.

```

// NgModuleFactoryLoader that uses SystemJS to load NgModuleFactory
@Injectable()
export class SystemJsNgModuleLoader implements NgModuleFactoryLoader {
    private _config: SystemJsNgModuleLoaderConfig;
}

```

Its constructor takes a `_compiler` as an optional parameter.

```

constructor(private _compiler: Compiler, @Optional() config?:
SystemJsNgModuleLoaderConfig) {
    this._config = config || DEFAULT_CONFIG;
}

```

In its `load()` method, if `_compiler` was provided, it calls `loadAndCompile()`, otherwise it calls `loadFactory()`.

```

load(path: string): Promise<NgModuleFactory<any>> {
    const offlineMode = this._compiler instanceof Compiler;
    return offlineMode ? this.loadFactory(path) : this.loadAndCompile(path);
}

```

It is within `loadAndCompile()` that `_compiler.compileAppModuleAsync()` is called.

```

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

```



```

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

```

The `ng_module_factory.ts` file defines the `NgModuleRef` and `NgModuleInjector` abstract classes and the `AppModuleFactory` concrete class.

`NgModuleRef` is defined as:

```

/**
 * Represents an instance of an NgModule created via a NgModuleFactory.
 * `NgModuleRef` provides access to the NgModule Instance as well other
 * objects related to this NgModule Instance.
 */
export abstract class NgModuleRef<T> {
  abstract get injector(): Injector;
  abstract get componentFactoryResolver(): ComponentFactoryResolver;
  abstract get instance(): T;
  abstract destroy(): void;
  abstract onDestroy(callback: () => void): void;
}

```

`NgModuleFactory` is used to create an `NgModuleRef` instance:

```

export abstract class NgModuleFactory<T> {
  abstract get moduleType(): Type<T>;
  abstract create(parentInjector: Injector|null): NgModuleRef<T>;
}

```

Core/View Feature (core/src/view)

The Core/View feature provides view management functionality. In Angular terminology, a view is an area of screen that can display content and where user events can be detected. A view is a bundle of elements. Views can be arranged hierarchically via the use of View Containers.

This comment from:

- [<ANGULAR-MASTER>/packages/core/src/linker/view_ref.ts](https://github.com/angular/angular/blob/master/packages/core/src/linker/view_ref.ts)

explains the relationship between elements and views:

“A View is a fundamental building block of the application UI. It is the smallest grouping of elements which are created and destroyed together.

Properties of elements in a View can change, but the structure (number and order) of elements in a View cannot. Changing the structure of Elements can only be done by inserting, moving or removing nested Views via a ViewContainerRef. Each View can contain many View Containers.”

One could say views are the heart of an Angular application. Core/View is a substantial feature that represents over a third of the entire Core package source tree.

Private Exports API

The Core/View feature is not intended to be used directly by Angular applications. Core/View does not export anything as part of the public API to the Core Package. As we have already discussed, [core/index.ts](#) describes the public API for Core, and its one line content exports [core/public_api.ts](#), which in turn exports [core/src/core.ts](#) and this exports nothing from the view sub-directory (searching for “view” results in no matches).

Some Core/Linker public exports are of classes which are used as base classes for types implemented in Core/View. In the file:

- [<ANGULAR-MASTER>/packages/core/src/linker.ts](#)

Your attention is drawn to these lines:

```
export {TemplateRef} from './linker/template_ref';
export {ViewContainerRef} from './linker/view_container_ref';
export {EmbeddedViewRef, ViewRef} from './linker/view_ref';
```

Core/View is intended to be used internally by other Angular packages. Hence we see its exports being defined as part of these two private export files:

- [<ANGULAR-MASTER>/packages/core/src/codegen_private_exports.ts](#)
- [<ANGULAR-MASTER>/packages/core/src/core_private_export.ts](#)

The first of these has these view-related exports:

```
export {
  clearOverrides as eclearOverrides,
  overrideComponentView as eoverrideComponentView,
  overrideProvider as eoverrideProvider
} from './view/index';
export {
  NOT_FOUND_CHECK_ONLY_ELEMENT_INJECTOR as
  eNOT_FOUND_CHECK_ONLY_ELEMENT_INJECTOR
} from './view/provider';
```

The second has these:

```
export {ArgumentType as eArgumentType,
  BindingFlags as eBindingFlags, DepFlags as eDepFlags,
  EMPTY_ARRAY as eEMPTY_ARRAY, EMPTY_MAP as eEMPTY_MAP,
  NodeFlags as eNodeFlags, QueryBindingType as eQueryBindingType,
  QueryValueType as eQueryValueType, ViewDefinition as eViewDefinition,
  ViewFlags as eViewFlags, anchorDef as eand,
  createComponentFactory as eccf, createNgModuleFactory as ecmf,
  createRendererType2 as ecrt, directiveDef as edid, elementDef as eeld,
  elementEventFullName as eelementEventFullName,
  getComponentViewDefinitionFactory as egetComponentViewDefinitionFactory,
  inlineInterpolate as einlineInterpolate, interpolate as einterpolate,
  moduleDef as emod, moduleProvideDef as empd, ngContentDef as encd,
  nodeValue as enov, pipeDef as epid, providerDef as eprd,
  pureArrayDef as epad, pureObjectDef as epod, purePipeDef as eppd,
  queryDef as equd, textDef as eted, unwrapValue as eunv, viewDef as evid
} from './view/index';
```

Core/View Implementation

The source file:

- [<ANGULAR-MASTER>/packages/core/src/view/types.ts](#)

defines helper types used elsewhere in Core/View. Let's start with `ViewContainerData` which simply extends `ViewContainerRef` and adds an internal property to record view data for embedded views:

```
export interface ViewContainerData extends ViewContainerRef {
  _embeddedViews: ViewData[];
}
```

The `TemplateData` interface extends `TemplateRef` and just adds an array of `ViewData` instances:

```
export interface TemplateData extends TemplateRef<any> {
  // views that have been created from the template
  // of this element, but inserted into the embeddedViews of
  // another element. By default, this is undefined.
  _projectedViews: ViewData[];
}
```

The source file:

- [<ANGULAR-MASTER>/packages/core/src/view/refs.ts](#)

provides internal implementations of references (refs).

`ViewContainerRef_` (note the trailing `_`) implements `ViewContainerData` which in turn implements the public `ViewContainerRef`.

It takes an `ElementData` for its anchor element in its constructor:

```
class ViewContainerRef_ implements ViewContainerData {
  _embeddedViews: ViewData[] = [];
  constructor(private _view: ViewData,
               private _elDef: NodeDef,
               private _data: ElementData) {}
  get element(): ElementRef {
    return new ElementRef(this._data.renderElement);
  }
  get injector(): Injector { return new Injector_(this._view, this._elDef); }
  get parentInjector(): Injector { .. }
  get(index: number): ViewRef|null {.. }
  get length(): number { return this._embeddedViews.length; }
  createEmbeddedView<C>(templateRef: TemplateRef<C>, context?: C,
                       index?: number): EmbeddedViewRef<C> {.. }
  createComponent<C>(
    componentFactory: ComponentFactory<C>, index?: number,
    injector?: Injector, projectableNodes?: any[][],
    ngModuleRef?: NgModuleRef<any>): ComponentRef<C> { .. }
  insert(viewRef: ViewRef, index?: number): ViewRef { .. }
  move(viewRef: ViewRef_, currentIndex: number): ViewRef {..}
  clear(): void {..}
}
```

The two create methods are implemented as follows:

```
createEmbeddedView<C>(templateRef: TemplateRef<C>,
                     context?: C, index?: number): EmbeddedViewRef<C> {
1   const viewRef = templateRef.createEmbeddedView(context || <any>{});
2   this.insert(viewRef, index);
   return viewRef;
}
```

```

    }

    createComponent<C>(
      componentFactory: ComponentFactory<C>,
      index?: number,
      injector?: Injector,
      projectableNodes?: any[][][],
      ngModuleRef?: NgModuleRef<any>): ComponentRef<C> {
      const contextInjector = injector || this.parentInjector;
      if (!ngModuleRef && !
        (componentFactory instanceof ComponentFactoryBoundToModule)) {
        ngModuleRef = contextInjector.get(NgModuleRef);
      }
      const componentRef =
3       componentFactory.create(
          contextInjector, projectableNodes, undefined, ngModuleRef);
4       this.insert(componentRef.hostView, index);
      return componentRef;
    }

```

We see the difference between them – `createEmbeddedView()` calls **1** the `TemplateRef`'s `createEmbeddedView()` method and inserts **2** the resulting `ViewRef`; whereas `createComponent()` calls **3** the component factory's `create` method, and with the resulting `ComponentRef`, inserts **4** its `HostView`. Note the return type is different for each create method – the first returns an `EmbeddedViewRef` whereas the second returns a `ComponentRef`.

The implementations of `insert`, `indexOf`, `remove` and `detach` result in use of appropriate view management APIs:

```

insert(viewRef: ViewRef, index?: number): ViewRef {
  const viewRef_ = <ViewRef_>viewRef;
  const viewData = viewRef._view;
  attachEmbeddedView(this._view, this._data, index, viewData);
  viewRef_.attachToViewContainerRef(this);
  return viewRef;
}

move(viewRef: ViewRef_, currentIndex: number): ViewRef {
  const previousIndex = this._embeddedViews.indexOf(viewRef._view);
  moveEmbeddedView(this._data, previousIndex, currentIndex);
  return viewRef;
}

indexOf(viewRef: ViewRef): number {
  return this._embeddedViews.indexOf((<ViewRef_>viewRef)._view);
}

remove(index?: number): void {
  const viewData = detachEmbeddedView(this._data, index);
  if (viewData) {
    Services.destroyView(viewData);
  }
}

detach(index?: number): ViewRef|null {
  const view = detachEmbeddedView(this._data, index);
  return view ? new ViewRef_(view) : null;
}

```

The `ComponentRef_` concrete class extends `ComponentRef`. Its constructor takes in an `ViewRef`, which is used in the `destroy` method and to set the component's change

detector ref and host view:

```

class ComponentRef_ extends ComponentRef<any> {
    public readonly hostView: ViewRef;
    public readonly instance: any;
    public readonly changeDetectorRef: ChangeDetectorRef;
    private _elDef: NodeDef;

    constructor(
        private _view: ViewData,
        private _viewRef: ViewRef,
        private _component: any) {
        super();
        this._elDef = this._view.def.nodes[0];
        this.hostView = _viewRef;
        this.changeDetectorRef = _viewRef;
        this.instance = _component;
    }
    get location(): ElementRef {
        return new ElementRef(asElementData(
            this._view, this._elDef.nodeIndex).renderElement);
    }
    get injector(): Injector { return new Injector_(this._view, this._elDef); }
    get componentType(): Type<any> { return <any>this._component.constructor; }

    destroy(): void { this._viewRef.destroy(); }
    onDestroy(callback: Function): void { this._viewRef.onDestroy(callback); }
}

```

The TemplateRef_ implementation has a constructor that takes a ViewData for the parent and a NodeDef. Its createEmbeddedView method returns a new ViewRef_ based on those parameters.

```

class TemplateRef_ extends TemplateRef<any> implements TemplateData {
    _projectedViews: ViewData[];
    constructor(private _parentView: ViewData, private _def: NodeDef) {
        super(); }

    createEmbeddedView(context: any): EmbeddedViewRef<any> {
        return new ViewRef_(Services.createEmbeddedView(
            this._parentView, this._def,
            this._def.element!.template!, context));
    }

    get elementRef(): ElementRef {
        return new ElementRef(asElementData(
            this._parentView, this._def.nodeIndex).renderElement);
    }
}

```

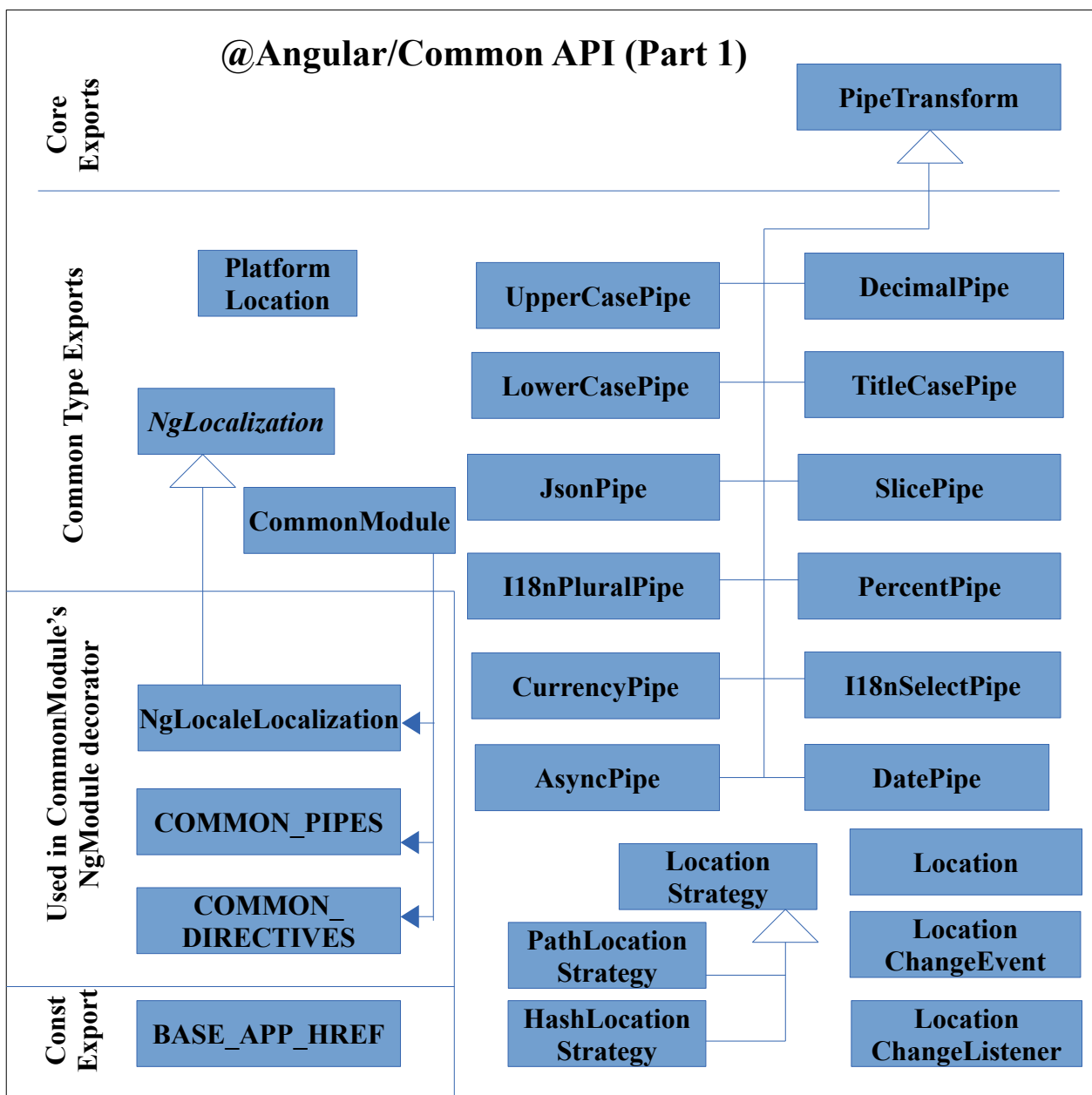
6: The Common Package

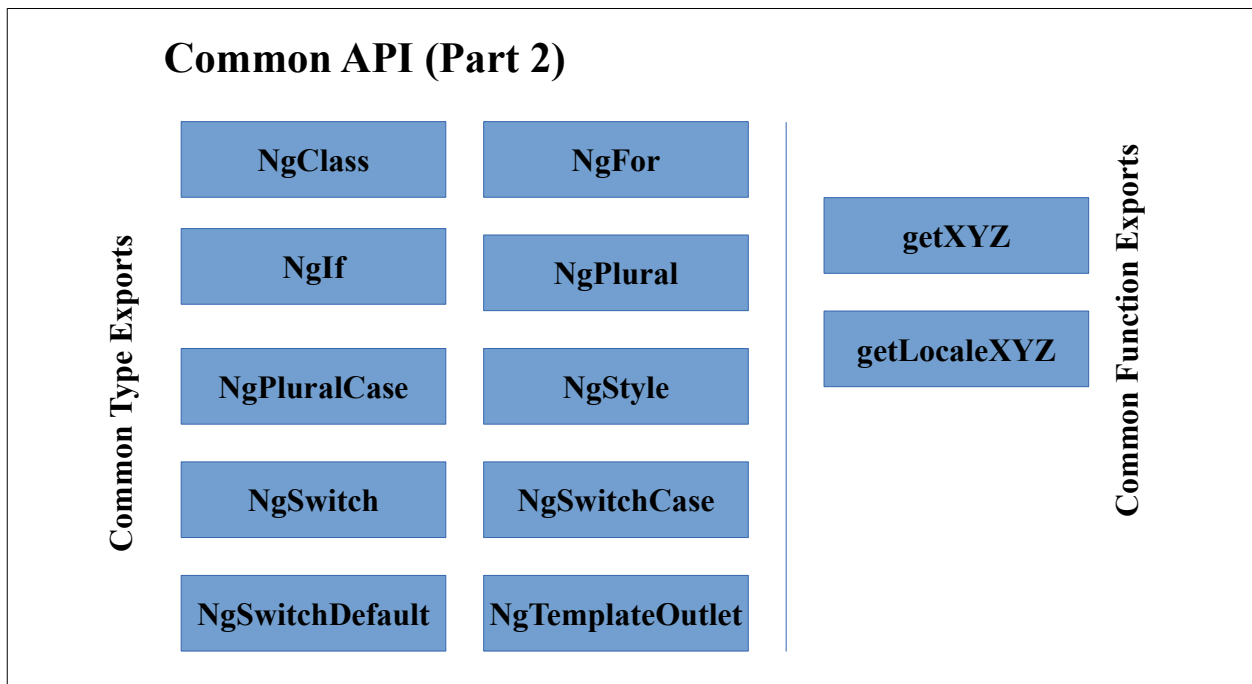
Overview

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

Common Public API

The public API of the Common package can be represented as:





The export API is defined by:

- [<ANGULAR-MASTER>/packages/common/index.ts](https://github.com/angular/angular/blob/master/packages/common/index.ts)

simply as:

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

which in turn is defined as:

```
/**
 * @module
 * @description
 * Entry point for all public APIs of this package.
 */
export * from './src/common';
```

```
// This file only reexports content of the `src` folder. Keep it that way.
```

The [src/common.ts](#) file contains:

```
export * from './location/index';
export {NgLocaleLocalization, NgLocalization} from './i18n/localization';
export {registerLocaleData} from './i18n/locale_data';
export {Plural, NumberFormatStyle, FormStyle, Time, TranslationWidth,
  FormatWidth, NumberSymbol, WeekDay, getCurrencySymbol, getLocaleDayPeriods,
  getLocaleDayNames, getLocaleMonthNames, getLocaleId, getLocaleEraNames,
  getLocaleWeekEndRange, getLocaleFirstDayOfWeek, getLocaleDateFormat,
  getLocaleDateTimeFormat, getLocaleExtraDayPeriodRules,
  getLocaleExtraDayPeriods, getLocalePluralCase, getLocaleTimeFormat,
  getLocaleNumberSymbol, getLocaleNumberFormat, getLocaleCurrencyName,
  getLocaleCurrencySymbol} from './i18n/locale_data_api';
export {parseCookieValue as eparseCookieValue} from './cookie';
export {CommonModule, DeprecatedI18NPipesModule} from './common_module';
export {NgClass, NgForOf, NgForOfContext, NgIf, NgIfContext, NgPlural,
  NgPluralCase, NgStyle, NgSwitch, NgSwitchCase, NgSwitchDefault,
  NgTemplateOutlet, NgComponentOutlet} from './directives/index';
export {DOCUMENT} from './dom_tokens';
export {AsyncPipe, DatePipe, I18nPluralPipe, I18nSelectPipe, JsonPipe,
  LowerCasePipe, CurrencyPipe, DecimalPipe, PercentPipe, SlicePipe,
  UpperCasePipe, TitleCasePipe} from './pipes/index';
export {PLATFORM_BROWSER_ID as ePLATFORM_BROWSER_ID,
  PLATFORM_SERVER_ID as ePLATFORM_SERVER_ID,
  PLATFORM_WORKER_APP_ID as ePLATFORM_WORKER_APP_ID,
  PLATFORM_WORKER_UI_ID as ePLATFORM_WORKER_UI_ID,
  isPlatformBrowser, isPlatformServer, isPlatformWorkerApp,
  isPlatformWorkerUi} from './platform_id';
export {VERSION} from './version';
```

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

The exported location types are listed in:

- [<ANGULAR-MASTER>/packages/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 interfaces. 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/guide/router>

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:

- http

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

and these files:

- BUILD.bazel
- index.ts
- package.json
- public_api.ts
- rollup.config.js
- tsconfig-build.json

The tsconfig-build.json content is:

```
{
  "extends": "../tsconfig-build.json",
  "compilerOptions": {
    "rootDir": ".",
    "baseUrl": ".",
    "paths": {
      "@angular/core": ["../../dist/packages/core"]
    },
    "outDir": "../../dist/packages/common"
  },
  "files": [
    "public_api.ts",
    "../../node_modules/zone.js/dist/zone.js.d.ts"
  ],
  "angularCompilerOptions": {
    "annotateForClosureCompiler": true,
    "strictMetadataEmit": false,
    "skipTemplateCodegen": true,
    "flatModuleOutFile": "common.js",
    "flatModuleId": "@angular/common"
  }
}
```

It lists as files to build as public_api.ts and brings in zone.js.d.ts.

Source

common/src

The common/src directory has these sub-directories:

- directives
- i18n
- location
- pipes

and these source files:

- common_module.ts

- common.ts
- cookie.ts
- dom_tokens.ts
- platform_id.ts
- version.ts

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

```
// Note: This does not contain the location providers,
// as they need some platform specific implementations to work.
/**
 * The module that includes all the basic Angular directives
 * like {@link NgIf}, {@link NgForOf}, ...
 */
@NgModule({
  declarations: [COMMON_DIRECTIVES, COMMON_PIPES],
  exports: [COMMON_DIRECTIVES, COMMON_PIPES],
  providers: [
    {provide: NgLocalization, useClass: NgLocaleLocalization},
  ],
})
export class CommonModule { }
```

`dom_tokens.ts` defines the `DOCUMENT` const (note the important comment):

```
/**
 * 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 Web Worker).
 */
export const DOCUMENT = new InjectionToken<Document>('DocumentToken');
```

The `platform_id.ts` file contains these simple functions to determine which platform is in use:

```
export function isPlatformBrowser(platformId: Object): boolean {
  return platformId === PLATFORM_BROWSER_ID;
}
export function isPlatformServer(platformId: Object): boolean {
  return platformId === PLATFORM_SERVER_ID;
}
export function isPlatformWorkerApp(platformId: Object): boolean {
  return platformId === PLATFORM_WORKER_APP_ID;
}
export function isPlatformWorkerUi(platformId: Object): boolean {
  return platformId === PLATFORM_WORKER_UI_ID;
}
```

common/src/i18n

The `src/i18n/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, locale?: string): 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, locale?:
string): string {
  let key = `=${value}`;

  if (cases.indexOf(key) > -1) {
    return key;
  }
  key = ngLocalization.getPluralCategory(value, locale);

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

  if (cases.indexOf('other') > -1) {
    return 'other';
  }

  throw new Error(`No plural message found for value "${value}"`);
}
```

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

```
//Returns the plural case based on the local
@Injectable()
export class NgLocaleLocalization extends NgLocalization {
  constructor(
    @Inject(LOCALE_ID) protected locale: string,
    /** @deprecated from v5 */
    @Optional()
    @Inject(DEPRECATED_PLURAL_FN) protected deprecatedPluralFn?:
      ((locale: string, value: number|string) => Plural)|null) {
    super();
  }

  getPluralCategory(value: any, locale?: string): string {
    const plural =
      this.deprecatedPluralFn ? this.deprecatedPluralFn(
        locale || this.locale, value) :
      getLocalePluralCase(locale || this.locale)(value);
  }
}
```

common/src/directives

This directory has the following source files:

- `index.ts`
- `ng_class.ts`
- `ng_component_outlet.ts`
- `ng_for_of.ts`

- `ng_if.ts`
- `ng_plural.ts`
- `ng_style.ts`
- `ng_switch.ts`
- `ng_template_outlet.ts`

The `index.ts` file exports the directive types:

```
export {
  NgClass,
  NgComponentOutlet,
  NgForOf,
  NgForOfContext,
  NgIf,
  NgIfContext,
  NgPlural,
  NgPluralCase,
  NgStyle,
  NgSwitch,
  NgSwitchCase,
  NgSwitchDefault,
  NgTemplateOutlet
};
```

along with a definition for `COMMON_DIRECTIVES`, which is:

```
/**
 * A collection of Angular directives that are likely to be used
 * in each and every Angular application.
 */
export const COMMON_DIRECTIVES: Provider[] = [
  NgClass,
  NgComponentOutlet,
  NgForOf,
  NgIf,
  NgTemplateOutlet,
  NgStyle,
  NgSwitch,
  NgSwitchCase,
  NgSwitchDefault,
  NgPlural,
  NgPluralCase,
];
```

The various `ng_` files implement the directives. Let's 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 = null;
  private _elseTemplateRef: TemplateRef<NgIfContext> | null = null;
  private _thenViewRef: EmbeddedViewRef<NgIfContext> | null = null;
  private _elseViewRef: EmbeddedViewRef<NgIfContext> | null = 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 some setters defined as input properties, for the variations of if:

```

@Input()
set ngIf(condition: any) {
  this._context.$implicit = this._context.ngIf = 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 =
1          this._viewContainer.createEmbeddedView(
              this._thenTemplateRef, this._context);
      }
    }
  } else {
    if (!this._elseViewRef) {
      this._viewContainer.clear();
      this._thenViewRef = null;
      if (this._elseTemplateRef) {
        this._elseViewRef =
2          this._viewContainer.createEmbeddedView(
              this._elseTemplateRef, this._context);
      }
    }
  }
}

```

The important calls here are (**1** & **2**) 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_of.ts](#), which implements the `NgFor`

class (and a helper class - NgForOfContext). The helper class is implemented as:

```
export class NgForOfContext<T> {
  constructor(
    public $implicit: T, public ngForOf: NgIterable<T>,
    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; }
}
```

NgFor is defined as:

```
@Directive({selector: '[ngFor][ngForOf]'})
export class NgForOf<T> implements DoCheck, OnChanges {

  private _differ: IterableDiffer<T>|null = null;
  private _trackByFn: TrackByFunction<T>;

  constructor(
    private _viewContainer: ViewContainerRef, private _template:
    TemplateRef<NgForOfContext<T>>,
    private _differs: IterableDiffers) {}
}
```

The first thing to note about NgFor's implementation is the class implements `DoCheck` and `OnChanges` lifecycle. The `DoCheck` class is a lifecycle hook defined in `@angular/core/src/metadata/lifecycle_hooks.ts` as:

```
export interface DoCheck { ngDoCheck(): void; }
```

`OnChanges` is defined in the same file as:

```
export interface OnChanges { 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 sub-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 = new InjectionToken<string>('appBaseHref');
```

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

```

@Inject()
export class HashLocationStrategy extends LocationStrategy {
  private _baseHref: string = '';
  constructor(
    private _platformLocation: PlatformLocation,
    @Optional() @Inject(APP_BASE_HREF) _baseHref?: string) {
    super();
    if (_baseHref != null) {
      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. They have quite distinct purposes.

```

export abstract class PlatformLocation { }
  abstract getBaseHrefFromDOM(): string;
  abstract onPopState(fn: LocationChangeListener): void;
  abstract onHashChange(fn: LocationChangeListener): void;

  abstract get pathname(): string;
  abstract get search(): string;
  abstract get hash(): string;

  abstract replaceState(state: any, title: string, url: string): void;

  abstract pushState(state: any, title: string, url: string): void;

  abstract forward(): void;

  abstract back(): void;
}

```

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 a parameter:

```
@Injectable()
export class Location {
  _subject: EventEmitter<any> = new EventEmitter();
  _baseHref: string;
  _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 to the platform's `popState` events.
subscribe(
  onNext: (value: PopStateEvent) => void,
  onThrow?: ((exception: any) => void)|null,
  onReturn?: (() => void)|null): ISubscription {
  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. All the pipes implement the `PipeTransform` interface. 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 (value == null) return value;

  if (!this.supports(value)) {
    throw 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 function invalidPipeArgumentError(type: Type<any>, value: Object) {
  return Error(
    `InvalidPipeArgument: '${value}' for pipe '${stringify(type)}'`); }
}
```

7: The Common/HTTP Sub-package

The Common/Http sub-package 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.

Note that in earlier versions of Angular, HTTP was a top-level package (similar to core, common, router, forms) but in recent versions of Angular it has been moved to be a sub-package of Common. Be aware that in older books, blogs and code samples you will still find references to this older top-level HTTP package. You will also find its API still described in the Angular API documentation, with all its types marked as "deprecated". For new code you write it is recommended to use the Common/HTTP sub-package and that is what we describe here.

An interesting additional project called in-memory-web-api can be used for a test-friendly implementation. It is located here:

- <https://github.com/angular/in-memory-web-api>

This can work with the new Common/HTTP sub-package or the older HTTP top-level package. The link above describes how to set up each – make sure it is appropriate for your needs. A very important note on that page is:

Always import the `HttpClientInMemoryWebApiModule` after the `HttpClientModule` to ensure that the in-memory backend provider supersedes the Angular version.

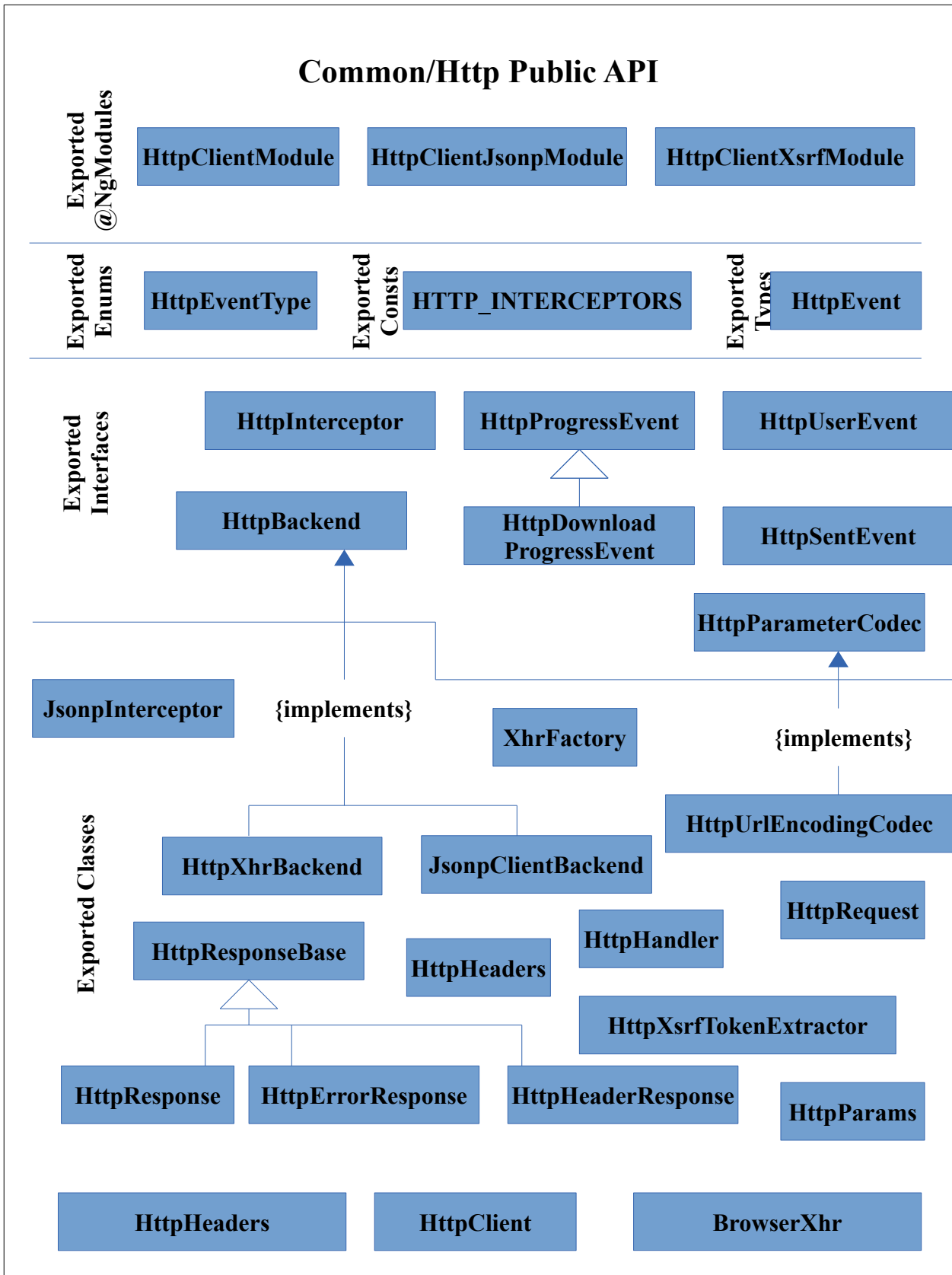
Common/Http Public API

The `common/http/src/index.ts` file exports [public_api.ts](#), which in turn defines the exports as:

```
export {HttpBackend, HttpClient} from './src/backend';

export {HttpClient} from './src/client';
export {HttpHeaders} from './src/headers';
export {HTTP_INTERCEPTORS, HttpInterceptor} from './src/interceptor';
export {JsonpClientBackend, JsonpInterceptor} from './src/jsonp';
export {HttpClientJsonpModule, HttpClientModule, HttpClientXsrfModule,
interceptingHandler as einterceptingHandler} from './src/module';
export {HttpParameterCodec, HttpParams, HttpUrlEncodingCodec} from
 './src/params';
export {HttpRequest} from './src/request';
export {HttpDownloadProgressEvent, HttpErrorResponse, HttpEvent,
HttpEventType, HttpHeadersResponse, HttpProgressEvent, HttpResponse,
HttpResponseBase, HttpSentEvent, HttpUserEvent} from './src/response';
export {HttpXhrBackend, XhrFactory} from './src/xhr';
export {HttpXsrfTokenExtractor} from './src/xsrf';
```

The exported public API of the Common/Http sub-package can be represented as:



Source Tree Layout

The source tree for the Common/Http sub-package contains these directories:

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

and these files:

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

http/src

The http/src directory has following source files:

- backend.ts
- client.ts
- headers.ts
- interceptors.ts
- jsonp.ts
- module.ts
- params.ts
- request.ts
- response.ts
- xhr.ts
- xsrf.ts

The module.ts file has a number of `NgModule` definitions, the main one being:

```
@NgModule({
  imports: [
    HttpClientXsrfModule.withOptions({
      cookieName: 'XSRF-TOKEN',
      headerName: 'X-XSRF-TOKEN',
    }),
  ],
  providers: [
    HttpClient,
    // HttpHandler is the backend + interceptors and is constructed
    // using the interceptingHandler factory function.
    {
      provide: HttpHandler,
      useFactory: interceptingHandler,
      deps: [HttpBackend, [new Optional()], new Inject(HTTP_INTERCEPTORS)],
    },
    HttpXhrBackend,
    {provide: HttpBackend, useExisting: HttpXhrBackend},
    BrowserXhr,
    {provide: XhrFactory, useExisting: BrowserXhr},
  ],
})
export class HttpClientModule { }
```

That `HttpXHRBackend` provider for `HttpClientModule` 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>

We note `HttpClientModule` uses `HttpClientXsrfModule` (we'll see where `XSRF_COOKIE_NAME` and `XSRF_HEADER_NAME` are defined when examining `XSRF` shortly):

```
@NgModule({
  providers: [
    HttpXsrfInterceptor,
    {provide: HTTP_INTERCEPTORS,
      useExisting: HttpXsrfInterceptor, multi: true},
    {provide: HttpXsrfTokenExtractor, useClass: HttpXsrfCookieExtractor},
    {provide: XSRF_COOKIE_NAME, useValue: 'XSRF-TOKEN'},
    {provide: XSRF_HEADER_NAME, useValue: 'X-XSRF-TOKEN'},
  ],
})
export class HttpClientXsrfModule {.. }
```

The `headers.ts` file provides the `HttpHeaders` class. This is essentially a wrapper around a `Map` data structure. It defines its primary data structure as:

```
private headers: Map<string, string[]>;
```

It offers functionality to work with that data structure.

```
// Immutable set of Http headers, with lazy parsing.
export class HttpHeaders {
  /**
   * Internal map of lowercase header names to values.
   */
  private headers: Map<string, string[]>;

  /**
   * Internal map of lowercased header names to the normalized
   * form of the name (the form seen first).
   */
  private normalizedNames: Map<string, string> = new Map();

  /**
   * Complete the lazy initialization of this object (needed before reading).
   */
  private lazyInit: HttpHeaders|Function|null;

  /**
   * Queued updates to be materialized the next initialization.
   */
  private lazyUpdate: Update[]|null = null;

  constructor(headers?: string|{[name: string]: string | string[]}) {.. }
}
```

The `request.ts` file implements the `Request` class. A request is a request method, a set of headers, and a content type. Depending on the request method, a body may or

may not be needed. To supply initialization parameters, this interface is defined:

```
interface HttpRequestInit {
  headers?: HttpHeaders;
  reportProgress?: boolean;
  params?: HttpParams;
  responseType?: 'arraybuffer'|'blob'|'json'|'text';
  withCredentials?: boolean;
}
```

We see its use in the constructor for Request:

```
// Next, need to figure out which argument holds the HttpRequestInit
// options, if any.
let options: HttpRequestInit|undefined;

// Check whether a body argument is expected. The only valid way to omit
// the body argument is to use a known no-body method like GET.
if (mightHaveBody(this.method) || !!fourth) {
  // Body is the third argument, options are the fourth.
  this.body = (third !== undefined) ? third as T : null;
  options = fourth;
} else {
  // No body required, options are the third argument.
  // The body stays null.
  options = third as HttpRequestInit;
}

// If options have been passed, interpret them.
if (options) {...}
```

Similarly, the response.ts implements the Response classes, which are based on the HttpResponseBase class:

```
export abstract class HttpResponseBase {
  // All response headers.
  readonly headers: HttpHeaders;
  // Response status code.
  readonly status: number;
  // Textual description of response status code.
  readonly statusText: string;
  //URL of the resource retrieved, or null if not available.
  readonly url: string|null;
  // Whether the status code falls in the 2xx range.
  readonly ok: boolean;
  // Type of the response, narrowed to either the full response
  // or the header.
  readonly type: HttpEventType.Response|HttpEventType.ResponseHeader;
}
```

The backend.ts file provides classes for pluggable connection handling. By providing alternative implementations of these, flexible server communication is supported. Note what the comments in the code say:

```
/**
 * Transforms an `HttpRequest` into a stream of `HttpEvent`s, one of
 * which will likely be a * `HttpResponse`.
 *
 * `HttpHandler` is injectable. When injected, the handler instance
 * dispatches requests to the
 * first interceptor in the chain, which dispatches to the second, etc,
```

```

    * eventually reaching the `HttpBackend`.
    *
    * In an `HttpInterceptor`, the `HttpHandler` parameter is the
    * next interceptor in the chain.
    */
export abstract class HttpHandler {
  abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;
}

/**
 * A final `HttpHandler` which will dispatch the request via browser HTTP
 * APIs to a backend.
 *
 * Interceptors sit between the `HttpClient` interface and the `HttpBackend`.
 *
 * When injected, `HttpBackend` dispatches requests directly to the backend,
 * without going through the interceptor chain.
 */
export abstract class HttpBackend implements HttpHandler {
  abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;
}

```

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.

When communicating with a real remote server, the main workload is performed by `HttpXhrBackend` class which is defined in [xhr.ts](#):

```

@Injectable()
export class HttpXhrBackend implements HttpBackend {
  constructor(private xhrFactory: XhrFactory) {}
}

```

The [xsrf.ts](#) file defines two injectable classes that helps with XSRF (Cross Site Request Forgery) protection:

```

// `HttpXsrfTokenExtractor` which retrieves the token from a cookie.
@Injectable()
export class HttpXsrfCookieExtractor implements HttpXsrfTokenExtractor {
  private lastCookieString: string = '';
  private lastToken: string|null = null;

  constructor(
    @Inject(DOCUMENT) private doc: any,
    @Inject(PLATFORM_ID) private platform: string,
    @Inject(XSRF_COOKIE_NAME) private cookieName: string) {}

  getToken(): string|null ..
}

// `HttpInterceptor` which adds an XSRF token to eligible outgoing requests.
@Injectable()
export class HttpXsrfInterceptor implements HttpInterceptor {
  constructor(
    private tokenService: HttpXsrfTokenExtractor,
    @Inject(XSRF_HEADER_NAME) private headerName: string) {}
}

```

```
    intercept(req: HttpRequest<any>, next: HttpHandler):  
Observable<HttpEvent<any>> {..  
    }  
}
```

This files also defines:

```
export const XSRF_COOKIE_NAME =  
    new InjectionToken<string>('XSRF_COOKIE_NAME');  
export const XSRF_HEADER_NAME =  
    new InjectionToken<string>('XSRF_HEADER_NAME');
```

These are used when defining the `@NgModule HttpClientXsrfModule` as we saw earlier.

8: The Platform-Browser Package

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 (web workers), 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 various platform packages – the main ones are defined [here in the Platform-Browser](#) package. Note that these render 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 packages:

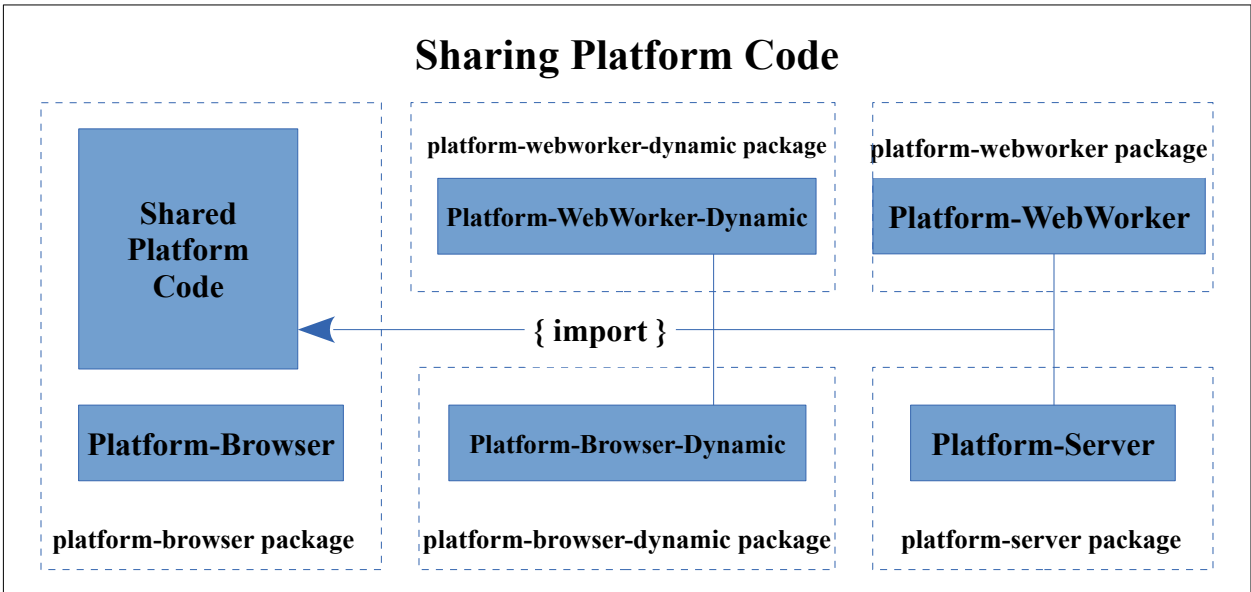
- platform-browser (runs in the browser’s main UI thread and uses the offline template compiler),
- platform-browser-dynamic (runs in the browser’s main UI thread and uses the runtime template compiler),
- platform-webworker (runs in a web worker and uses the offline compiler),
- platform-webworker-dynamic (runs in a web worker and uses the runtime template compiler) and
- platform-server (runs in the server and can use either the offline or runtime template compiler)

Shared functionality relating to platforms is in Platform-Browser and imported by the other platform packages. So Platform-Browser is a much bigger package compared to the other platform packages. In this chapter we will explore Platform-Browser and we will cover the others in the subsequent chapters.

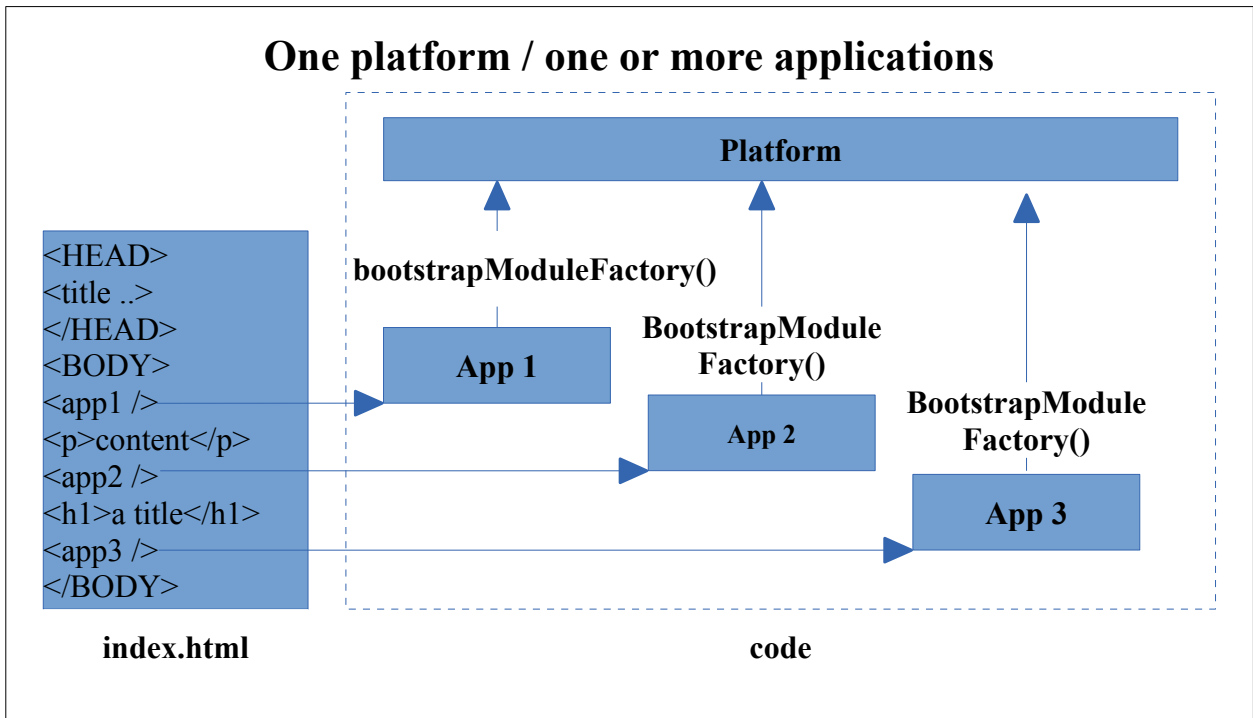
Platform-Browser is how application code can interact with the browser when running in the main UI thread and assumes the offline template compiler has been used to pre-generate a module factory. For production use, platform-browser is likely to be the platform of choice, as it results in the fastest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

The Platform-Browser package does not depend on the Compiler package since it assumes compilation has occurred Ahead-Of-Time (AOT) using Compiler-CLI, the command-line interface that wraps the Compiler package. In contrast, Platform-Browser-Dynamic does import directly from the Compiler package – for example, see:

- [<ANGULAR-MASTER>/packages/platform-browser-dynamic/src/compiler_factory.ts](#)



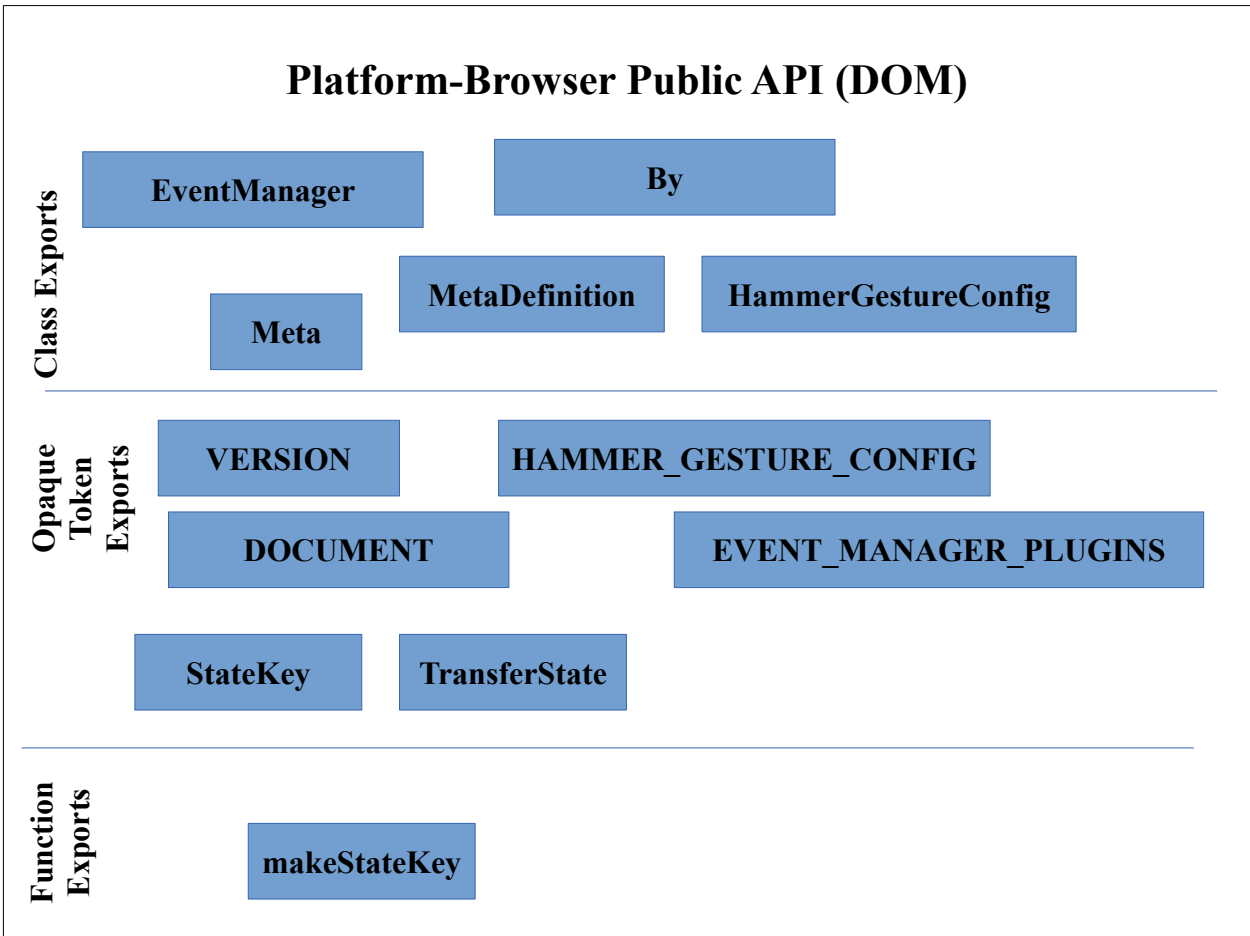
There is exactly one platform instance per thread (main browser UI thread or web worker). Multiple applications may run in the same thread, and they interact with the same platform instance.



Platform-Browser Public API

The platform-browser API can be sub-divided into these functional areas: browser-related, DOM and security.

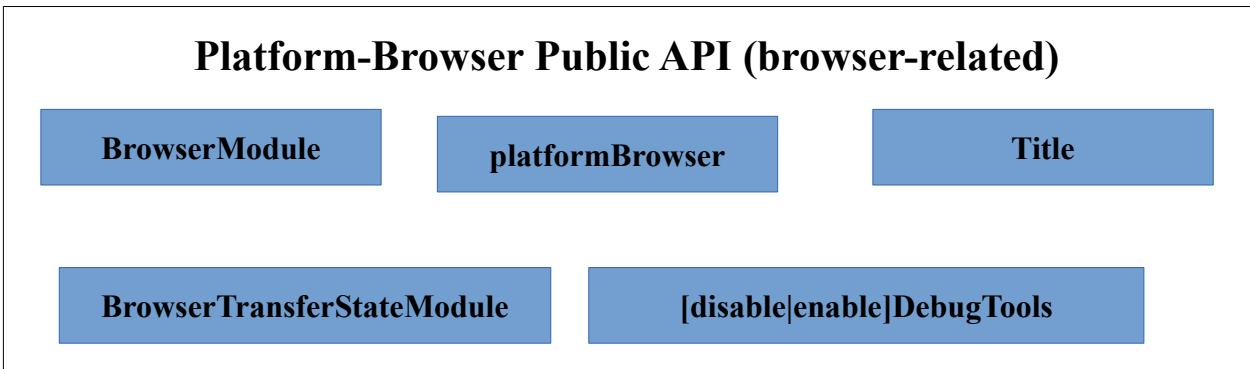
The DOM related API can be represented as:



Hammer is for touch input. `EventManager` handles the flow of events.

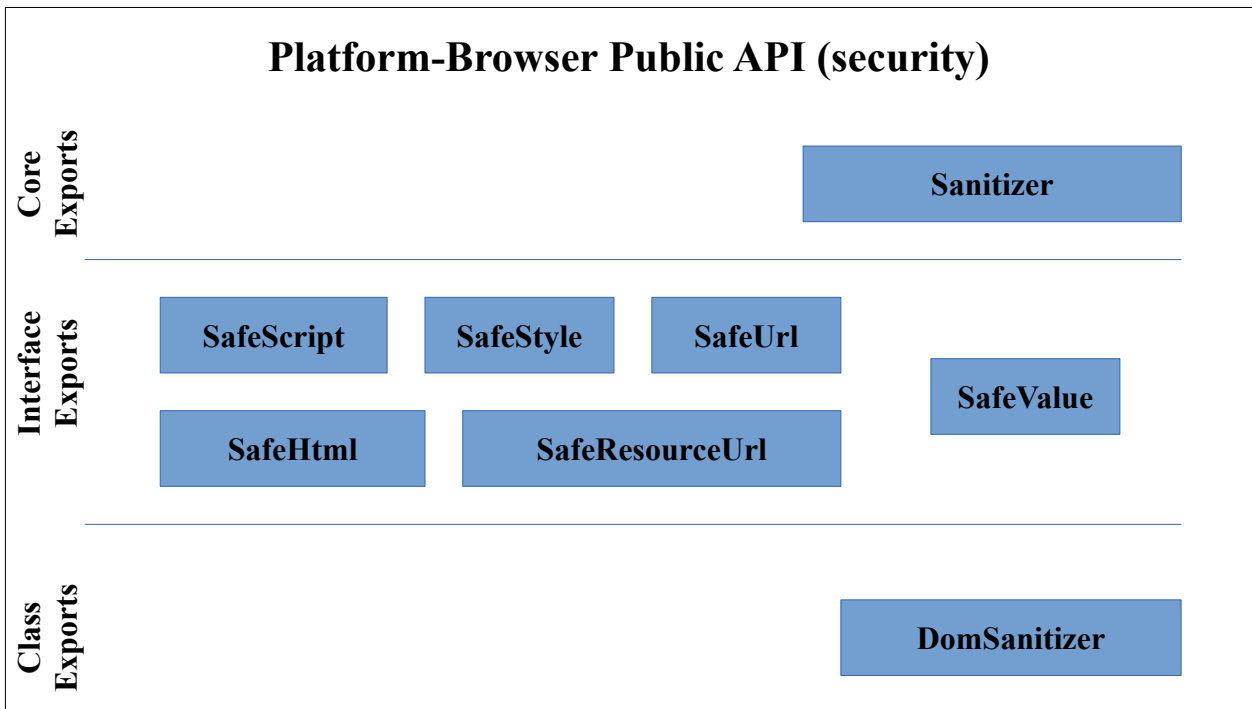
Some important classes – e.g. `DomAdapter` - are not publically exported. Instead, they are registered with the dependency injector and that is how they are made available to application code (we will shortly examine both in detail).

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 package has:

- index.ts

which has the single line:

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

and this in turn has:

```
/**
 * @module
 * @description
 * Entry point for all public APIs of this package.
 */
export * from './src/platform-browser';
// This file only reexports content of the `src` folder. Keep it that way.
```

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

```
export {BrowserModule, platformBrowser} from './browser';
export {Meta, MetaDefinition} from './browser/meta';
export {Title} from './browser/title';
export {disableDebugTools, enableDebugTools} from './browser/tools/tools';
export {BrowserTransferStateModule, StateKey, TransferState, makeStateKey}
  from './browser/transfer_state';
export {By} from './dom/debug/by';
export {EVENT_MANAGER_PLUGINS, EventManager}
  from './dom/events/event_manager';
export {HAMMER_GESTURE_CONFIG, HAMMER_LOADER, HammerGestureConfig,
  HammerLoader} from './dom/events/hammer_gestures';
export {DomSanitizer, SafeHtml, SafeResourceUrl, SafeScript, SafeStyle,
  SafeUrl, SafeValue} from './security/dom_sanitization_service';
```

```
export * from './private_export';
export {VERSION} from './version';
// This must be exported so it doesn't get tree-shaken away prematurely
export {ELEMENT_PROBE_PROVIDERS__POST_R3__ as
  eELEMENT_PROBE_PROVIDERS__POST_R3__} from './dom/debug/ng_probe';
```

The `private_export.ts` file exports types that are intended to be used by other Angular packages and not be normal Angular applications. It has this content:

```
export {BROWSER_SANITIZATION_PROVIDERS as eBROWSER_SANITIZATION_PROVIDERS,
  INTERNAL_BROWSER_PLATFORM_PROVIDERS as eINTERNAL_BROWSER_PLATFORM_PROVIDERS,
  initDomAdapter as einitDomAdapter} from './browser';
export {BrowserDomAdapter as eBrowserDomAdapter}
  from './browser/browser_adapter';
export {BrowserPlatformLocation as eBrowserPlatformLocation}
  from './browser/location/browser_platform_location';
export {TRANSITION_ID as eTRANSITION_ID} from './browser/server-transition';
export {BrowserGetTestability as eBrowserGetTestability}
  from './browser/testability';
export {escapeHtml as eescapeHtml} from './browser/transfer_state';
export {ELEMENT_PROBE_PROVIDERS as eELEMENT_PROBE_PROVIDERS}
  from './dom/debug/ng_probe';
export {DomAdapter as eDomAdapter, getDOM as egetDOM,
  setRootDomAdapter as esetRootDomAdapter} from './dom/dom_adapter';
export {DomRendererFactory2 as eDomRendererFactory2,
  NAMESPACE_URI as eNAMESPACE_URI, flattenStyles as eflattenStyles,
  shimContentAttribute as eshimContentAttribute,
  shimHostAttribute as eshimHostAttribute} from './dom/dom_renderer';
export {DomEventsPlugin as eDomEventsPlugin} from './dom/events/dom_events';
export {HammerGesturesPlugin as eHammerGesturesPlugin}
  from './dom/events/hammer_gestures';
export {KeyEventsPlugin as eKeyEventsPlugin} from './dom/events/key_events';
export {DomSharedStylesHost as eDomSharedStylesHost,
  SharedStylesHost as eSharedStylesHost} from './dom/shared_styles_host';
export {DomSanitizerImpl as eDomSanitizerImpl}
  from './security/dom_sanitization_service';
```

Source Tree Layout

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

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

and these files in the root directory:

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

Source

platform-browser/src

This directory contains these sub-directories:

- browser
- dom
- security

and this file:

- browser.ts

The browser.ts file defines providers (`INTERNAL_BROWSER_PLATFORM_PROVIDERS` and `BROWSER_SANITIZATION_PROVIDERS`) and the NgModule-decorated `BrowserModule`, all important in the bootstrapping of an Angular application.

`INTERNAL_BROWSER_PLATFORM_PROVIDERS` is used to define platform-related providers for dependency injection:

```
export const INTERNAL_BROWSER_PLATFORM_PROVIDERS: StaticProvider[] = [
  {provide: PLATFORM_ID, useValue: PLATFORM_BROWSER_ID},
  {provide: PLATFORM_INITIALIZER, useValue: initDomAdapter, multi: true},
  {provide: PlatformLocation, useClass: BrowserPlatformLocation,
    deps: [DOCUMENT]},
  {provide: DOCUMENT, useFactory: _document, deps: []},
];
```

Sanitization providers mitigate XSS risks – note the comment:

```
/**
 * @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: StaticProvider[] = [
  {provide: Sanitizer, useExisting: DomSanitizer},
  {provide: DomSanitizer, useClass: DomSanitizerImpl, deps: [DOCUMENT]},
];
```

It is used in the definition of `BROWSER_MODULE_PROVIDERS`:

```
export const BROWSER_MODULE_PROVIDERS: StaticProvider[] = [
  BROWSER_SANITIZATION_PROVIDERS,
  {provide: APP_ROOT, useValue: true},
  {provide: ErrorHandler, useFactory: errorHandler, deps: []},
  {
    provide: EVENT_MANAGER_PLUGINS,
    useClass: DomEventsPlugin,
    multi: true,
    deps: [DOCUMENT, NgZone, PLATFORM_ID]
  },
  {provide: EVENT_MANAGER_PLUGINS, useClass: KeyEventsPlugin, multi: true,
  deps: [DOCUMENT]},
  {
    provide: EVENT_MANAGER_PLUGINS,
    useClass: HammerGesturesPlugin,
    multi: true,
```

```

        deps: [DOCUMENT, HAMMER_GESTURE_CONFIG, Console,
              [new Optional(), HAMMER_LOADER]]
    },
    {provide: HAMMER_GESTURE_CONFIG, useClass: HammerGestureConfig, deps: []},
    {
        provide: DomRendererFactory2,
        useClass: DomRendererFactory2,
        deps: [EventManager, DomSharedStylesHost, APP_ID]
    },
    {provide: RendererFactory2, useExisting: DomRendererFactory2},
    {provide: SharedStylesHost, useExisting: DomSharedStylesHost},
    {provide: DomSharedStylesHost, useClass: DomSharedStylesHost, deps:
[DOCUMENT]},
    {provide: Testability, useClass: Testability, deps: [NgZone]},
    {provide: EventManager, useClass: EventManager, deps:
[EVENT_MANAGER_PLUGINS, NgZone]},
    ELEMENT_PROBE_PROVIDERS,
];

```

This in turn is used in the definition of `BrowserModule`:

```

/**
 * Exports required infrastructure for all Angular apps.
 * Included by default in all Angular apps created with the CLI
 * `new` command.
 * Re-exports `CommonModule` and `ApplicationModule`, making their
 * exports and providers available to all apps.
 *
 * @publicApi
 */
@NgModule({providers: BROWSER_MODULE_PROVIDERS, exports: [CommonModule,
ApplicationModule]})
export class BrowserModule {
  constructor(@Optional() @SkipSelf() @Inject(BrowserModule) parentModule:
BrowserModule|null) {
    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).

```

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

```

As we saw, this function is used in an initializer by the app provider list:

```

export const INTERNAL_BROWSER_PLATFORM_PROVIDERS: StaticProvider[] = [
  {provide: PLATFORM_INITIALIZER, useValue: initDomAdapter, multi: true,
  ..};

```

This in turn is used by the call to `createPlatformFactory`:

```
export const platformBrowser: (extraProviders?: StaticProvider[]) =>
PlatformRef =
  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 application 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:

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

shared_styles_host.ts manages a set of styles.

Utils.ts contains simple string helpers:

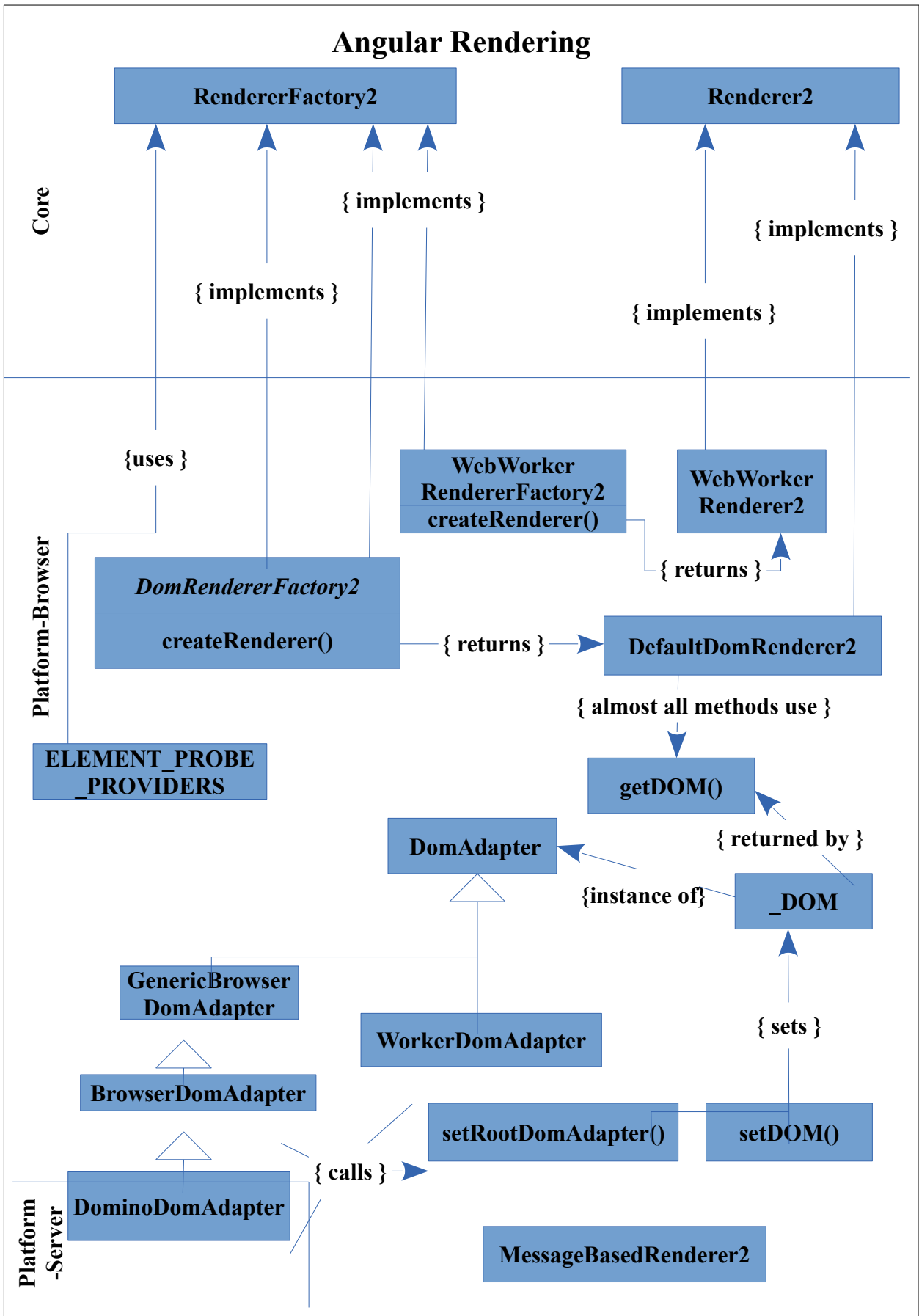
```
export function camelCaseToDashCase(input: string): string {
  return input.replace(CAMEL_CASE_REGEXP,
    (...m: string[]) => '-' + m[1].toLowerCase());
}

export function dashCaseToCamelCase(input: string): string {
  return input.replace(DASH_CASE_REGEXP,
    (...m: string[]) => 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 DefaultDomRenderer2.



For server applications with Platform-Server, a specialist DOM adapter called `DominoAdapter` will be used, and this results in content being written to an HTML file. `DominoAdapter` is implemented in the Platform-Server package here:

- [<ANGULAR-MASTER>/packages/platform-server/src/domino_adapter.ts](https://github.com/ANGULAR-MASTER/packages/platform-server/src/domino_adapter.ts)

It is based in this project:

- <https://github.com/fgnass/domino>

We'll explore this further when examining the Platform-Server package.

For more advanced browser applications that will use web workers, things are a little more complicated and involves these main classes: `WorkerRenderer2`, `MessageBasedRender2` 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 web worker and main browser UI thread. `WorkerRenderer2` runs in the web worker and forward all rendering calls over the message broker to the main browser UI thread, where an instance of `MessageBasedRender2` (which, despite its name, does not implement Core's renderer API) receives them and calls the regular `DefaultDomRenderer2`. We will shortly examine in detail how rendering works with web workers. We'll explore rendering and web workers in the later chapter on Platform-WebWorker.

When trying to figure out how the DOM adaptor works, the best place to start is:

- [<ANGULAR-MASTER>/packages/platform-browser/src/dom_adapter.ts](https://github.com/ANGULAR-MASTER/packages/platform-browser/src/dom_adapter.ts)

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

```
let _DOM: DomAdapter = null !;

export function getDOM() {
  return _DOM;
}

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

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

The `DomAdapter` class is a very long list of methods similar to what you would find a normal DOM API (pay attention to the security warning!)– here is just a sampling:

```
/**
 * 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 querySelector(el: any, selector: string): any;
  abstract querySelectorAll(el: any, selector: string): any[];
  abstract nodeName(node: any): string;
  abstract nodeValue(node: any): string|null;
  abstract parentElement(el: any): Node|null;
  abstract appendChild(el: any, node: any): any;
  abstract removeChild(el: any, node: any): any;
  abstract replaceChild(el: any, newNode: any, oldNode: any): any;
  abstract remove(el: any): Node;
  abstract insertBefore(parent: any, ref: any, node: any): any;
  abstract insertAfter(parent: any, el: any, node: any): any;
  abstract getText(el: any): string|null;
  abstract setText(el: any, value: string): any;
  abstract getValue(el: any): string;
  abstract setValue(el: any, value: string): any;
  abstract createComment(text: string): any;
  abstract createTemplate(html: any): HTMLElement;
  abstract createElement(tagName: any, doc?: any): HTMLElement;
  abstract createTextNode(text: string, doc?: any): Text;
  abstract removeStyle(element: any, styleName: string): any;
  abstract getStyle(element: any, styleName: string): string;
  abstract hasAttribute(element: any, attribute: string): boolean;
  abstract getAttribute(element: any, attribute: string): string|null;
  abstract setAttribute(element: any, name: string, value: string): any;
  abstract getCookie(name: string): string|null;
  abstract setCookie(name: string, value: string): any;
  ..
}

```

The `dom_renderer.ts` file defines the DOM renderer that relies on `getDOM()` to supply a `DomAdapter`. The main class it supplies is `DomRendererFactory2` which implements `RendererFactory2`. We saw earlier how `DomRendererFactory2` is used in the `BROWSER_MODULE_PROVIDERS` declaration, as used by `NgModule`:

```

export const BROWSER_MODULE_PROVIDERS: StaticProvider[] = [
  ..
  {
    provide: DomRendererFactory2,
    useClass: DomRendererFactory2,
    deps: [EventManager, DomSharedStylesHost, APP_ID]
  },
  {provide: RendererFactory2, useExisting: DomRendererFactory2},
  ..
];

```

The `DomRendererFactory2` class manages a map of strings to `Renderer2` instances.

```

export class DomRendererFactory2 implements RendererFactory2 {
  private rendererByCompId = new Map<string, Renderer2>();
  private defaultRenderer: Renderer2;

  constructor(
    private eventManager: EventManager, private sharedStylesHost:
    DomSharedStylesHost,
    @Inject(APP_ID) private appId: string) {
    this.defaultRenderer = new DefaultDomRenderer2(eventManager);
  }
}

```

Its `createRenderer()` method performs a switch on the encapsulation type and

returns an appropriate `Renderer2`:

```
createRenderer(element: any, type: RendererType2|null): Renderer2 {
  if (!element || !type) {
    return this.defaultRenderer;
  }
  switch (type.encapsulation) {
    case ViewEncapsulation.Emulated: {
      let renderer = this.rendererByCompId.get(type.id);
      if (!renderer) {
        renderer = new EmulatedEncapsulationDomRenderer2(
          this.eventManager, this.sharedStylesHost, type, this.appId);
        this.rendererByCompId.set(type.id, renderer);
      }
      (<EmulatedEncapsulationDomRenderer2>renderer).applyToHost(element);
      return renderer;
    }
    case ViewEncapsulation.Native:
    case ViewEncapsulation.ShadowDom:
      return new ShadowDomRenderer(
        this.eventManager,
        this.sharedStylesHost, element, type);
    default: {
      if (!this.rendererByCompId.has(type.id)) {
        const styles = flattenStyles(type.id, type.styles, []);
        this.sharedStylesHost.addStyles(styles);
        this.rendererByCompId.set(type.id, this.defaultRenderer);
      }
      return this.defaultRenderer;
    }
  }
}
```

`DefaultDomRenderer2` is where rendering actually occurs. It implements `Renderer2`. As an example, let's look at its `createElement()` method:

```
createElement(name: string, namespace?: string): any {
  if (namespace) {
    // In cases where Ivy (not ViewEngine) is giving us the actual
    // namespace, the look up by key
    // will result in undefined, so we just return the namespace here.
    return document.createElementNS(
      NAMESPACE_URIS[namespace] || namespace, name);
  }

  return document.createElement(name);
}
```

DOM debugging is supported via:

- `src/dom/debug/by.ts`
- `src/dom/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`:

```
// Predicates for use with {@link DebugElement}'s query functions.
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 debugElement.nativeElement != null ?
      getDOM().elementMatches(debugElement.nativeElement, selector) :
      false;
  };
}

// Match elements that have the given directive present.
static directive(type: Type<any>): Predicate<DebugElement> {
  return (debugElement) =>
    debugElement.providerTokens !.indexOf(type) !== -1;
}
}

```

DOM events are supported via:

- `src/dom/events/dom_events.ts`
- `src/dom/events/event_manager.ts`
- `src/dom/events/hammer_common.ts`
- `src/dom/events/hammer_gestures.ts`
- `src/dom/events/key_events.ts`

`event_manager.ts` provides two classes – `EventManager` and `EventManagerPlugin`.

`EventManager` manages an array of `EventManagerPlugins`, which is defined as:

```

export abstract class EventManagerPlugin {
  constructor(private _doc: any) {}

  manager !: EventManager;

  abstract supports(eventName: string): boolean;

  abstract addEventListener(
    element: HTMLElement, eventName: string, handler: Function): Function;

  addGlobalEventListener(
    element: string, eventName: string, handler: Function): Function {
    const target: HTMLElement = getDOM().getGlobalEventTarget(
      this._doc, element);

    if (!target) {
      throw new Error(
        `Unsupported event target ${target} for event ${eventName}`);
    }
    return this.addEventListener(target, eventName, handler);
  }
}

```

It provides two methods to add event listeners to targets represented either by an `HTMLElement` or a string. `EventManager` itself is defined as an injectable class:

```

/**
 * An injectable service that provides event management for Angular
 * through a browser plug-in.
 *
 * @publicApi

```

```

*/
@Injectable()
export class EventManager {
  private _plugins: EventManagerPlugin[];
  private _eventNameToPlugin = new Map<string, EventManagerPlugin>();

  /**
   * Initializes an instance of the event-manager service.
   */
  constructor(@Inject(EVENT_MANAGER_PLUGINS) plugins: EventManagerPlugin[],
private _zone: NgZone) {
    plugins.forEach(p => p.manager = this);
    this._plugins = plugins.slice().reverse();
  }

  /**
   * Registers a handler for a specific element and event.
   *
   * @param element The HTML element to receive event notifications.
   * @param eventName The name of the event to listen for.
   * @param handler A function to call when the notification occurs. Receives
the
   * event object as an argument.
   * @returns A callback function that can be used to remove the handler.
   */
  addEventListener(element: HTMLElement, eventName: string, handler:
Function): Function {
    const plugin = this._findPluginFor(eventName);
    return plugin.addEventListener(element, eventName, handler);
  }

  /**
   * Registers a global handler for an event in a target view.
   *
   * @param target A target for global event notifications. One of "window",
"document", or "body".
   * @param eventName The name of the event to listen for.
   * @param handler A function to call when the notification occurs. Receives
the
   * event object as an argument.
   * @returns A callback function that can be used to remove the handler.
   */
  addGlobalEventListener(target: string, eventName: string, handler:
Function): Function {
    const plugin = this._findPluginFor(eventName);
    return plugin.addGlobalEventListener(target, eventName, handler);
  }

  /**
   * Retrieves the compilation zone in which event listeners are registered.
   */
  getZone(): NgZone { return this._zone; }

  /** @internal */
  _findPluginFor(eventName: string): EventManagerPlugin {
    const plugin = this._eventNameToPlugin.get(eventName);
    if (plugin) {
      return plugin;
    }
  }
}

```

```

    const plugins = this._plugins;
    for (let i = 0; i < plugins.length; i++) {
      const plugin = plugins[i];
      if (plugin.supports(eventName)) {
        this._eventNameToPlugin.set(eventName, plugin);
        return plugin;
      }
    }
    throw new Error(`No event manager plugin found for event ${eventName}`);
  }
}

```

Its constructor is defined 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; }
}

```

Touch events via the hammer project are supported via the `hammer_gesture.ts` file. The list of supported touch events are:

```

const EVENT_NAMES = {
  // 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 by HammerGesturesPlugin:

```

@Injectable()
export class HammerGesturesPlugin extends EventManagerPlugin {
  constructor(
    @Inject(DOCUMENT) doc: any,
    @Inject(HAMMER_GESTURE_CONFIG) private _config: HammerGestureConfig,
    private console: Console,
    @Optional() @Inject(HAMMER_LOADER) private loader?: HammerLoader|null)
  {
    super(doc);
  }
}

```

Its addEventListener method is implemented as:

```

addEventListener(element: HTMLElement, eventName: string, handler: Function):
Function {
  const zone = this.manager.getZone();
  eventName = eventName.toLowerCase();
  if (!(window as any).Hammer && this.loader) {
    let cancelRegistration = false;
    let deregister: Function = () => { cancelRegistration = true; };
    this.loader()
      ..
    return () => { deregister(); };
  }
  1 return zone.runOutsideAngular(() => {
    // Creating the manager bind events, must be done outside of angular
    const mc = this._config.buildHammer(element);
    const callback = function(eventObj: HammerInput) {
      zone.runGuarded(function() { handler(eventObj); });
    };
    mc.on(eventName, callback);
    return () => {
      mc.off(eventName, callback);
      // destroy mc to prevent memory leak
      if (typeof mc.destroy === 'function') {
        mc.destroy();
      }
    };
  });
}

```

Note the use of `zone.runOutsideAngular()` **1**. Also note it does not implement `addGlobalEventListener`. Its 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 {
    const mc = new Hammer(element);

    mc.get('pinch').set({enable: true});
    mc.get('rotate').set({enable: true});

    for (const 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 `BROWSER_MODULE_PROVIDERS`:

```
export const BROWSER_MODULE_PROVIDERS: StaticProvider[] = [
  {
    provide: EVENT_MANAGER_PLUGINS,
    useClass: DomEventsPlugin,
    multi: true,
    deps: [DOCUMENT, NgZone, PLATFORM_ID]
  },
  {provide: EVENT_MANAGER_PLUGINS, useClass: KeyEventsPlugin, multi: true,
    deps: [DOCUMENT]},
  {
    provide: EVENT_MANAGER_PLUGINS,
    useClass: HammerGesturesPlugin,
    multi: true,
    deps: [DOCUMENT, HAMMER_GESTURE_CONFIG, Console,
      [new Optional(), HAMMER_LOADER]]
  },
  {provide: HAMMER_GESTURE_CONFIG, useClass: HammerGestureConfig, deps: []},
  {
    provide: DomRendererFactory2,
    useClass: DomRendererFactory2,
    deps: [EventManager, DomSharedStylesHost, APP_ID]
  },
  {provide: EventManager, useClass: EventManager,
    deps: [EVENT_MANAGER_PLUGINS, NgZone]},
  ELEMENT_PROBE_PROVIDERS,
  ..
];
```

Platform-browser/src/browser

This directory contains these files:

- `browser_adapter.ts`
- `generic_browser_adapter.ts`
- `meta.ts`
- `server-transition.ts`
- `testability.ts`
- `title.ts`

- `transfer_state.ts`
- `location/browser_platform_location.ts`
- `location/history.ts`
- `tools/browser.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 typeof (<any>document.body).createShadowRoot === 'function';
}
```

`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(doc: Document): string { return doc.title; }
setTitle(doc: Document, newTitle: string) { doc.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.
 *
 * @publicApi
 */
@Injectable({providedIn: 'root', useFactory: createTitle, deps: []})
export class Title {
  constructor(@Inject(DOCUMENT) private _doc: any) {}
```

```

/**
 * Get the title of the current HTML document.
 */
getTitle(): string { return getDOM().getTitle(this._doc); }

/**
 * Set the title of the current HTML document.
 * @param newTitle
 */
setTitle(newTitle: string) { getDOM().setTitle(this._doc, newTitle); }
}

```

The tools sub-directory contains `tools.ts` and `common_tools.ts`. The `AngularProfiler` class is defined in `common_tools.ts`:

```

export class AngularProfiler {
  appRef: ApplicationRef;

  constructor(ref: ComponentRef<any>) {
    this.appRef = ref.injector.get(ApplicationRef);
  }

  timeChangeDetection(config: any): ChangeDetectionPerfRecord {
    ..
    return new ChangeDetectionPerfRecord(msPerTick, numTicks);
  }
}

```

The `timeChangeDetection` method measures performance.

`Tools.ts` defines two functions `enableDebugTools()` and `disableDebugTools()` to add **1** and remove **2** a `ng` property to global:

```

const PROFILER_GLOBAL_NAME = 'profiler';

/**
 * 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)
 * 1. Type `ng.` (usually the console will show auto-complete suggestion)
 * 1. Try the change detection profiler `ng.profiler.timeChangeDetection()`
 *    then hit Enter.
 *
 * @experimental All debugging apis are currently experimental.
 */
export function enableDebugTools<T>(ref: ComponentRef<T>): ComponentRef<T> {
1 exportNgVar(PROFILER_GLOBAL_NAME, new AngularProfiler(ref));
  return ref;
}

/**
 * Disables Angular tools.
 *
 * @experimental All debugging apis are currently experimental.
 */
export function disableDebugTools(): void {
2 exportNgVar(PROFILER_GLOBAL_NAME, null);
}

```

```
}

```

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
 * {@link Location}.
 */
@Injectable()
export class BrowserPlatformLocation extends PlatformLocation {
  public readonly location: Location;
  private _history: History;

  constructor(@Inject(DOCUMENT) private _doc: any) {
    super();
    this._init();
  }
  ..
}
```

This manages two private fields, `location` and `history`:

```
public readonly location !: Location;
private _history !: History;
```

If you are wondering what the `!` means, this StackOverflow answer:

- <https://stackoverflow.com/questions/50983838/what-does-mean-in-typescript/50984662>

is what you are looking for:

“Specifically, the operation `x!` produces a value of the type of `x` with `null` and `undefined` excluded.”

They are initialized with `getDOM()`:

```
constructor(@Inject(DOCUMENT) private _doc: any) {
  super();
  this._init();
}
// This is moved to its own method so that
// `MockPlatformLocationStrategy` can overwrite it
_init() {
  (this as {location: Location}).location = getDOM().getLocation();
  this._history = getDOM().getHistory();
}
```

`history.ts` contains this simple function:

```
export function supportsState(): boolean {
  return !!window.history.pushState;
}
```

Platform-browser/src/security

This directory contains this file:

- `dom_sanitization_service.ts`

Security sanitizers help prevent the use of dangerous constructs in HTML, CSS styles and URLs. Sanitizers are configured in the `BrowserModule` class via an `NgModule` setting. This is the relevant extract from `platform-browser/src/browser.ts`:

```
export const BROWSER_SANITIZATION_PROVIDERS: StaticProvider[] = [
  {provide: Sanitizer, useExisting: DomSanitizer},
  {provide: DomSanitizer, useClass: DomSanitizerImpl, deps: [DOCUMENT]},
];
```

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`:

- [<ANGULAR_MASTER>/packages/core/src/sanitization/security.ts](#)

Let's recall they are defined as:

```
export enum SecurityContext {
  NONE = 0,
  HTML = 1,
  STYLE = 2,
  SCRIPT = 3,
  URL = 4,
  RESOURCE_URL = 5,
}
export abstract class Sanitizer {
  abstract sanitize(
    context: SecurityContext, value: {}|string|null): string|null;
}
```

In `dom_sanitization_service.ts` 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 if 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 `

```

The `DomSanitizerImpl` injectable class is what is supplied to `NgModule`:

```

@Injectables()
export class DomSanitizerImpl extends DomSanitizer {
  constructor(@Inject(DOCUMENT) private _doc: any) { super(); }
}

```

It can be divided into three sections, the `checkNotSafeValue` method, the `sanitize` method and the `bypassSecurityTrustXYZ` methods. The `checkNotSafeValue` method throws an error if 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 parameter, 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: SafeValue|string|null): string|null {
    if (value == null) return null;
    switch (ctx) {
        case SecurityContext.NONE:
            return value as string;
        case SecurityContext.HTML:
            if (value instanceof SafeHtmlImpl)
                return value.changingThisBreaksApplicationSecurity;
            this.checkNotSafeValue(value, 'HTML');
            return _sanitizeHtml(this._doc, String(value));
        case SecurityContext.STYLE:
            if (value instanceof SafeStyleImpl)
                return value.changingThisBreaksApplicationSecurity;
            this.checkNotSafeValue(value, 'Style');
            return _sanitizeStyle(value as string);
        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); }
```

9: The Platform-Browser-Dynamic Package

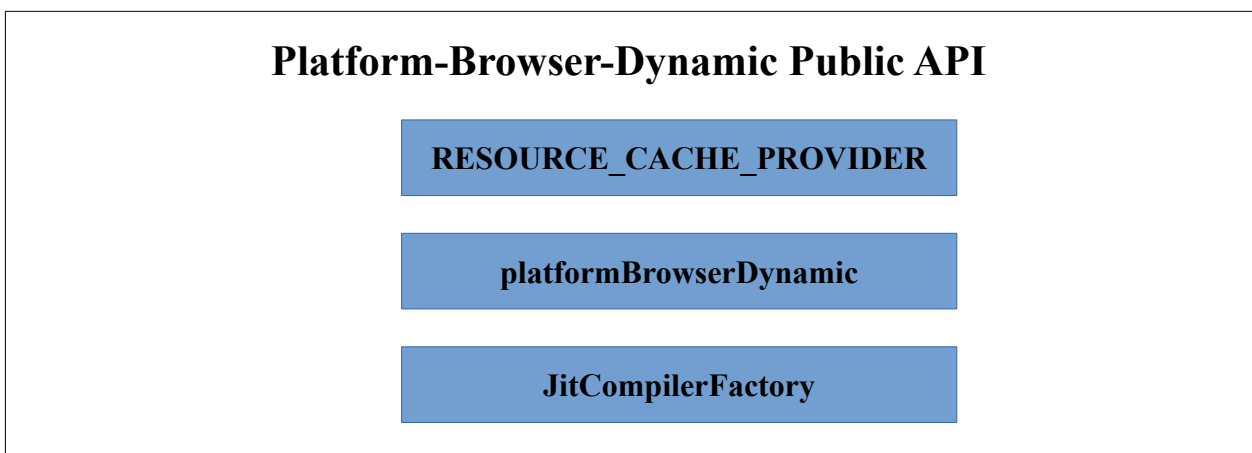
Overview

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 web worker, then the platform to use is `platformWorkerAppDynamic` from the Platform-WebWorker-Dynamic package (discussed in a later chapter).

Platform-Browser-Dynamic API

The exported Public API of the Platform-Browser-Dynamic package can be represented as:



It’s `index.ts` has just one line:

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

and the `public_api.ts` file has these lines:

```
/**
 * @module
 * @description
 * Entry point for all public APIs of this package.
 */
export * from './src/platform-browser-dynamic';
```

```
// This file only reexports content of the `src` folder. Keep it that way.
```

The `./src/platform-browser-dynamic.ts` file is where the exports are actually defined:


```
export * from './private_export';
export {VERSION} from './version';
export {JitCompilerFactory} from './compiler_factory';

/**
 * @publicApi
 */
export const RESOURCE_CACHE_PROVIDER: Provider[] =
  [{provide: ResourceLoader, useClass: CachedResourceLoader, deps: []}];

/**
 * @publicApi
 */
export const platformBrowserDynamic = createPlatformFactory(
  platformCoreDynamic, 'browserDynamic',
  INTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS);
```

The `platformBrowserDynamic` const 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`).

The `RESOURCE_CACHE_PROVIDER` const manages a cache of downloaded resources.

The `JitCompilerFactory` class creates a factory to access a just-in-time compiler.

The `src` directory for Platform-Browser-Dynamic also contains this private export file:

- [<ANGULAR-MASTER>/packages/platform-browser-dynamic/src/private_export.ts](#)

and it contains the following:

```
export {CompilerImpl as eCompilerImpl} from './compiler_factory';
export {platformCoreDynamic as eplatformCoreDynamic}
  from './platform_core_dynamic';
export {INTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS
  as eINTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS}
  from './platform_providers';
export {ResourceLoaderImpl as eResourceLoaderImpl}
  from './resource_loader/resource_loader_impl';
```

Source Tree Layout

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

- `src`
- `src/resource_loader`
- `test` (unit tests in Jasmine)
- `testing` (test tooling)

and these files in the root directory:

- `index.ts`
- `package.json`
- `public_api.ts`

Source

src

This directory contains these files:

- `compiler_factory.ts`
- `compiler_reflector.ts`
- `platform_core_dynamic.ts`
- `platform_providers.ts`

Let's start with [platform_providers.ts](#), which contains:

```
export const INTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS: StaticProvider[] =
[
  INTERNAL_BROWSER_PLATFORM_PROVIDERS,
  {
    provide: COMPILER_OPTIONS,
    useValue: {providers: [{provide: ResourceLoader,
                          useClass: ResourceLoaderImpl, deps: []}]},
    multi: true
  },
  {provide: PLATFORM_ID, useValue: PLATFORM_BROWSER_ID},
];
```

This extends `INTERNAL_BROWSER_PLATFORM_PROVIDERS` with two additional provider configurations. The first is `COMPILER_OPTIONS`, which itself needs a `ResourceLoader` (defined in the `Compiler` module), that we see is set to use `ResourceLoaderImpl` (defined in this package). The second is `PLATFORM_ID` which uses the value `PLATFORM_BROWSER_ID` (from the `Common` package).

This file:

- [ANGULAR-MASTER/packages/platform-browser-dynamic/src/compiler_factory.ts](#)

defines the `JitCompilerFactory` class.

Its `_mergeOptions` method takes a `CompilerOptions` parameter and returns a `CompilerOptions` instance with these additional entries:

```
function _mergeOptions(optionsArr: CompilerOptions[]): CompilerOptions {
  return {
    useJit: _lastDefined(optionsArr.map(
      options => options.useJit)),
    defaultEncapsulation: _lastDefined(optionsArr.map(
      options => options.defaultEncapsulation)),
    providers: _mergeArrays(optionsArr.map(
      options => options.providers !)),
    missingTranslation: _lastDefined(optionsArr.map(
      options => options.missingTranslation)),
    preserveWhitespaces: _lastDefined(optionsArr.map(
      options => options.preserveWhitespaces)),
  };
}
```

src/resource_loader

This directory has two files:

- `resource_loader_cache.ts`
- `resource_loader_impl.ts`

`CachedResourceLoader` is a cached version of a `ResourceLoader` (from the `Compiler` package). When the template compiler needs to access documents, it passes the job on to a configured resource loader. `ResourceLoader` is defined in:

- [<ANGULAR-MASTER>/packages/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>|string { return ''; }
}
```

`CachedResourceLoader` uses a promise wrapper to resolve or reject a get request:

```
/**
 * An implementation of ResourceLoader that uses a template cache to avoid
 * doing an actual ResourceLoader.
 *
 * The template cache needs to be built and loaded into window.$templateCache
 * via a separate mechanism.
 *
 * @publicApi
 */
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.');
```

`ResourceLoaderImpl` is a different injectable implementation of `ResourceLoader` that use `XMLHttpRequest()`:

```
@Injectable()
export class ResourceLoaderImpl extends ResourceLoader {
  get(url: string): Promise<string> {
    let resolve: (result: any) => void;
    let reject: (error: any) => void;
    const promise = new Promise<string>((res, rej) => {
      resolve = res;
      reject = rej;
    });
    const 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;
  }
}
```

10: The Platform-Server Package

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 (but for the curious, a library called Domino is used which provides a DOM for node apps).

Platform-Server API

Its index.ts is:

```
// This file is not used to build this module. It is only used during editing
// by the TypeScript language service and during build for verification.
// `ngc` replaces this file with production index.ts when it rewrites private
// symbol names.

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

Its public_api.ts file is:

```
/**export {
 * @module
 * @description
 * Entry point for all public APIs of this package.
 */
export * from './src/platform-server';
```

```
// This file only reexports content of the `src` folder. Keep it that way.
```

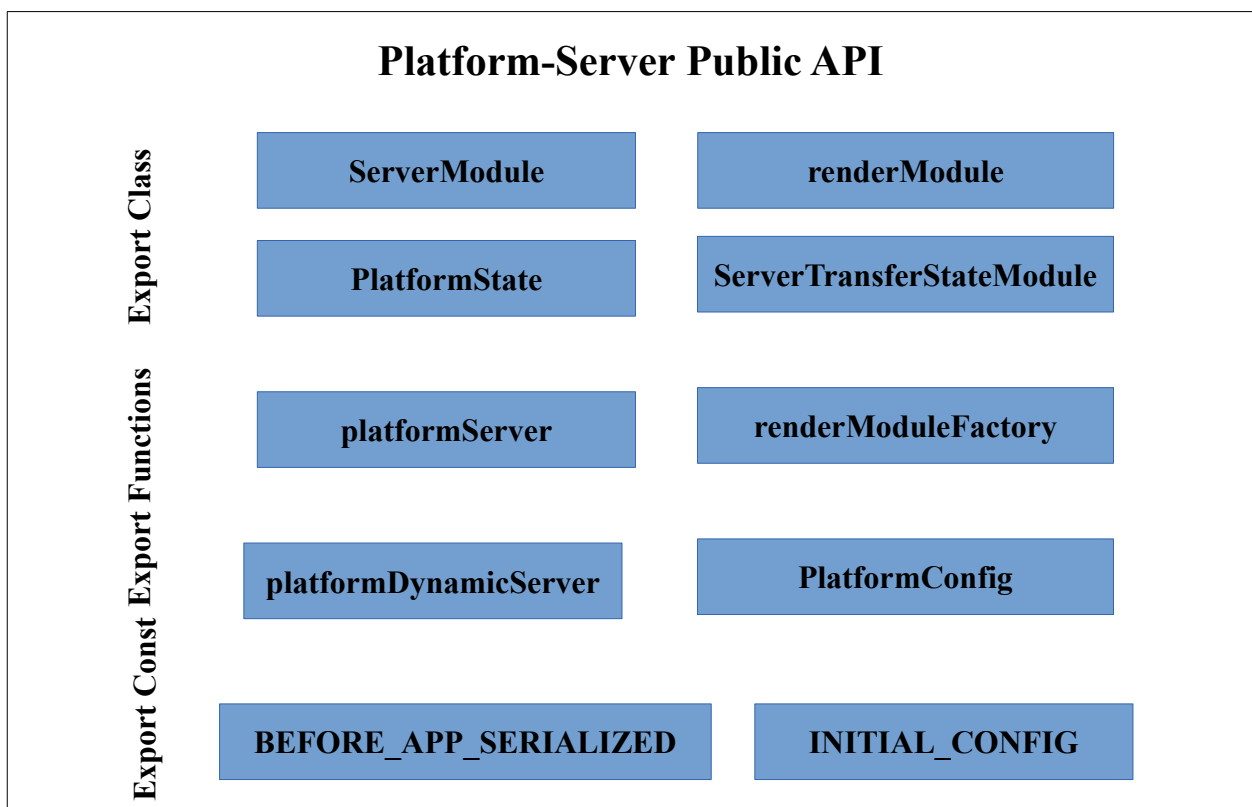
Its `src/platform-server.ts` file (that's a hyphen rather than an underscore) is:

```
export {PlatformState} from './platform_state';
export {ServerModule, platformDynamicServer, platformServer} from './server';
export {BEFORE_APP_SERIALIZED, INITIAL_CONFIG, PlatformConfig}
  from './tokens';
export {ServerTransferStateModule} from './transfer_state';
export {renderModule, renderModuleFactory} from './utils';
export * from './private_export';
export {VERSION} from './version';
```

The main `src` directory for Platform-Server also contains this `private_export.ts` file:

```
export {
  INTERNAL_SERVER_PLATFORM_PROVIDERS as eINTERNAL_SERVER_PLATFORM_PROVIDERS,
  SERVER_RENDER_PROVIDERS as eSERVER_RENDER_PROVIDERS} from './server';
export {ServerRendererFactory2 as eServerRendererFactory2}
  from './server_renderer';
```

The exported API of the Platform-Server package can be represented as:



`ServerModule` represents the `NgModule` for this module. `platformServer()` uses the offline compiler. `platformDynamicServer()` 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
- public_api.ts

Package.json has dependencies listed as:

```
"peerDependencies": {
  "@angular/animations": "0.0.0-PLACEHOLDER",
  "@angular/common": "0.0.0-PLACEHOLDER",
  "@angular/compiler": "0.0.0-PLACEHOLDER",
  "@angular/core": "0.0.0-PLACEHOLDER",
  "@angular/platform-browser": "0.0.0-PLACEHOLDER",
  "@angular/platform-browser-dynamic": "0.0.0-PLACEHOLDER"
},
"dependencies": {
  "domino": "^2.1.2",
  "tslib": "^1.9.0",
  "xhr2": "^0.1.4"
},
```

domino is the server-side DOM editing package that we need.

Source

platform-server/src

These source files are present:

- domino_adapter.ts
- http.ts
- location.ts
- platform_state.ts
- server_events.ts
- server_renderer.ts
- server.ts
- styles_host.ts
- tokens.ts
- transfer_state.ts
- utils.t
- version.ts

There are no sub-directories beneath src.

The server.ts file declares this const:

```
export const INTERNAL_SERVER_PLATFORM_PROVIDERS: StaticProvider[] = [
  {provide: DOCUMENT, useFactory: _document, deps: [Injector]},
  {provide: PLATFORM_ID, useValue: PLATFORM_SERVER_ID},
  {provide: PLATFORM_INITIALIZER, useFactory: initDominoAdapter, multi: true,
    deps: [Injector]}, {
    provide: PlatformLocation,
    useClass: ServerPlatformLocation,
```

```

    deps: [DOCUMENT, [Optional, INITIAL_CONFIG]]
  },
  {provide: PlatformState, deps: [DOCUMENT]},
  // Add special provider that allows multiple instances of
  // platformServer* to be created.
  {provide: ALLOW_MULTIPLE_PLATFORMS, useValue: true}
];

```

The factory functions `platformServer` and `platformDynamicServer` create server platforms that 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 domino adapter, in a call to the local function `initDominoAdapter()` which calls `makeCurrent()` for the `DominoAdapter`:

```

function initDominoAdapter(injector: Injector) {
  return () => { DominoAdapter.makeCurrent(); };
}

```

Secondly, it adds `PlatformLocation`, which is used by applications to interact with location (URL) information. It is set to a class defined in `location.ts`, `ServerPlatformLocation`, which mostly just throws exceptions:

```

/**
 * Server-side implementation of URL state. Implements `pathname`, `search`,
 * and `hash` but not the state stack.
 */
@Injectable()
export class ServerPlatformLocation implements PlatformLocation {
  public readonly href: string = '/';
  public readonly hostname: string = '/';
  public readonly protocol: string = '/';
  public readonly port: string = '/';
  public readonly pathname: string = '/';
  public readonly search: string = '';
  public readonly hash: string = '';
  private _hashUpdate = new Subject<LocationChangeEvent>();
  ..
  getBaseHrefFromDOM(): string { return getDOM().getBaseHref(this._doc) !; }

  onPopState(fn: LocationChangeListener): void {
    // No-op: a state stack is not implemented, so
    // no events will ever come.
  }

  onHashChange(fn: LocationChangeListener): void
  { this._hashUpdate.subscribe(fn); }

  get url(): string { return `${this.pathname}${this.search}${this.hash}`; }

  forward(): void { throw new Error('Not implemented'); }
  back(): void { throw new Error('Not implemented'); }

  // History API isn't available on server, therefore return undefined
  getState(): unknown { return undefined; }
}

```


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:

- [<ANGULAR-MASTER>/packages/core/src/platform_core_providers.ts](#)

as:

```
const _CORE_PLATFORM_PROVIDERS: StaticProvider[] = [
  // Set a default platform name for platforms that don't set it explicitly.
  {provide: PLATFORM_ID, useValue: 'unknown'},
  {provide: PlatformRef, deps: [Injector]},
  {provide: TestabilityRegistry, deps: []},
  {provide: Console, deps: []},
];

/**
 * This platform has to be included in any other platform
 *
 * @publicApi
 */
export const platformCore = createPlatformFactory(
  null, 'core', _CORE_PLATFORM_PROVIDERS);
```

platformCoreDynamic adds additional provider config (for the dynamic compiler) to platformCore and is defined in:

- [<ANGULAR-MASTER>/packages/platform-browser-dynamic/src/platform-browser-dynamic.ts](#)

as:

```
export const platformCoreDynamic = createPlatformFactory(
  platformCore, 'coreDynamic', [
    {provide: COMPILER_OPTIONS, useValue: {}, multi: true},
    {provide: CompilerFactory, useClass: JitCompilerFactory,
      deps: [COMPILER_OPTIONS]},
  ]);
```

createPlatformFactory() is defined in:

- [<ANGULAR-MASTER>/packages/core/src/application_ref.ts](#)

it calls Core's createPlatform() with the supplied parameters, resulting in a new platform being constructed.

The remain part of Platform-Server's server.ts file to discuss is the definition of the NgModule called ServerModule:

```
/**
 * The ng module for the server.
 *
 * @publicApi
 */
```

```

@NgModule({
  exports: [BrowserModule],
  imports: [HttpClientModule, NoopAnimationsModule],
  providers: [
    SERVER_RENDER_PROVIDERS,
    SERVER_HTTP_PROVIDERS,
    {provide: Testability, useValue: null},
    {provide: ViewportScroller, useClass: NullViewportScroller},
  ],
})
export class ServerModule {
}

function _document(injector: Injector) {
  let config: PlatformConfig|null = injector.get(INITIAL_CONFIG, null);
  if (config && config.document) {
    return parseDocument(config.document, config.url);
  } else {
    return getDOM().createHtmlDocument();
  }
}

```

The `domino_adapter.ts` file create a DOM adapter for Domino. The Domino library is a DOM engine for HTML5 that runs in Node. Its project page is:

- <https://github.com/fgnass/domino>

and states:

“As the name might suggest, domino's goal is to provide a DOM in Node.”

The `domino_adapter.ts` file provides an adapter class, `DominoDomAdapter`, based on Domino’s serialization functionality that implements a `DomAdapter` suitable for use in server environments.

The `domino_adapter.ts` file has these functions to parse and serialize a document:

```

/**
 * Parses a document string to a Document object.
 */
export function parseDocument(html: string, url = '/') {
  let window = domino.createWindow(html, url);
  let doc = window.document;
  return doc;
}

/**
 * Serializes a document to string.
 */
export function serializeDocument(doc: Document): string {
  return (doc as any).serialize();
}

```

We see `serializeDocument` being called from:

- [<ANGULAR-MASTER>/packages/platform-server/src/platform_state.ts](https://github.com/angular/angular/blob/master/packages/platform-server/src/platform_state.ts)

as:

```

@Injectable()
export class PlatformState {
  constructor(@Inject(DOCUMENT) private _doc: any) {}

  /**
   * Renders the current state of the platform to string.
   */
  renderToString(): string { return serializeDocument(this._doc); }

  /**
   * Returns the current DOM state.
   */
  getDocument(): any { return this._doc; }
}

```

BrowserDomAdapter is defined in:

- [<ANGULAR-MASTER>/packages/platform-browser/src/browser/browser_adapter.ts](https://github.com/angular/angular/blob/master/packages/platform-browser/src/browser/browser_adapter.ts)

DominoAdapter simply extends BrowserDomAdapter:

```

/**
 * DOM Adapter for the server platform based on
 * https://github.com/fgnass/domino.
 */
export class DominoAdapter extends BrowserDomAdapter {
  private static defaultDoc: Document;
}

```

Its static `makeCurrent()` method, that we saw Universal Angular uses for server-side rendering, initializes those three variables and then calls `setRootDomAdapter()`:

```

static makeCurrent() {
  setDomTypes();
  setRootDomAdapter(new DominoAdapter());
}

```

Recall that `setRootDomAdapter()` is defined in:

- [<ANGULAR-MASTER>/packages/platform-browser/src/dom/dom_adapter.ts](https://github.com/angular/angular/blob/master/packages/platform-browser/src/dom/dom_adapter.ts)

as:

```

let _DOM: DomAdapter = null !;

export function getDOM() {
  return _DOM;
}

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

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

```

and that `getDOM()` is used by the `DOMRenderer`. Hence our `DominoAdapter` gets wired into the DOM renderer.

Many of the DOM adapter methods throw exceptions as they do not make sense server-side:

```
getHistory(): History { throw _notImplemented('getHistory'); }  
getLocation(): Location { throw _notImplemented('getLocation'); }  
getUserAgent(): string { return 'Fake user agent'; }
```

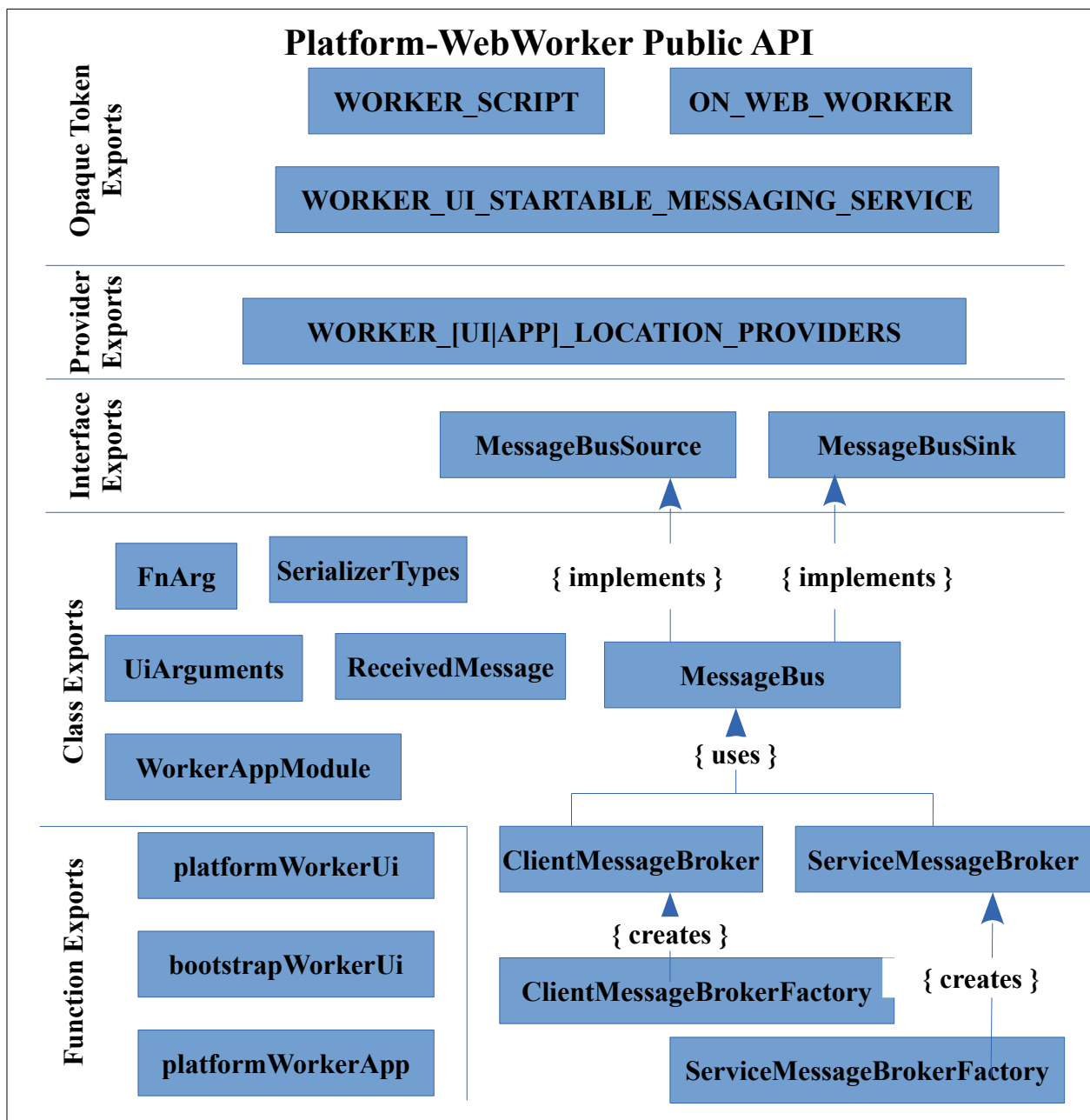
11: The Platform-WebWorker Package

Overview

Platform-WebWorker enables applications run inside a web worker.

Platform-WebWorker Public API

The Platform-WebWorker Public API is large and can be represented as:



The `index.ts` file is simply:

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

and `public_api.ts` has this content:

```
/**
 * @module
 * @description
 * Entry point for all public APIs of this package.
 */
export * from './src/platform-webworker';
// This file only reexports content of the `src` folder. Keep it that way.
```

The `src/platform-webworker.ts` file lists the various exports and defines the `bootstrapWorkerUi` function:

```
export {VERSION} from './version';
export {ClientMessageBroker, ClientMessageBrokerFactory, FnArg, UiArguments}
  from './web_workers/shared/client_message_broker';
export {MessageBus, MessageBusSink, MessageBusSource}
  from './web_workers/shared/message_bus';
export {SerializerTypes} 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: StaticProvider[] = []): Promise<PlatformRef> {
  // For now, just creates the worker ui platform...
  const platform = platformWorkerUi([
    {provide: WORKER_SCRIPT, useValue: workerScriptUri},
    ...customProviders,
  ]);

  return Promise.resolve(platform);
}
```

Source Tree Layout

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

- `src`
- `test` (unit tests in Jasmine)

(there is no testing directory) and these files:

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

Source

platform-webworker/src

In addition to the 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.

A web worker cannot use the DOM directly in a web browser. Therefore Angular's Platform-WebWorker creates a message bus between the web worker and the main UI thread and rendering (and event processing) takes place over this message bus.

The platformWorkerApp const creates a platform factory for a worker app:

```
export const platformWorkerApp = createPlatformFactory(
  platformCore, 'workerApp', [{provide: PLATFORM_ID, useValue:
  PLATFORM_WORKER_APP_ID}]);
```

Two important helper functions are supplied. createMessageBus creates the message bus that will supply communications between the main UI thread and the web worker:

```
export function createMessageBus(zone: NgZone): MessageBus {
  const sink = new PostMessageBusSink(_postMessage);
  const source = new PostMessageBusSource();
  const bus = new PostMessageBus(sink, source);
  bus.attachToZone(zone);
  return bus;
}
```

setupWebWorker makes the web worker'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 web workers. 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 no underlying DOM API in a web worker. So for an app running in a web worker, 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:

```
// The ng module for the worker app side.
@NgModule({
  providers: [
    BROWSER_SANITIZATION_PROVIDERS,
    Serializer,
    {provide: DOCUMENT, useValue: null},
    ClientMessageBrokerFactory,
    ServiceMessageBrokerFactory,
```

```

    WebWorkerRendererFactory2,
    {provide: RendererFactory2, useExisting: WebWorkerRendererFactory2},
    {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: StaticProvider[] = [
  {provide: NgZone, useFactory: createNgZone, deps: []},
  {
    provide: MessageBasedRenderer2,
    deps: [ServiceMessageBrokerFactory, MessageBus,
      Serializer, RenderStore, RendererFactory2]
  },
  {provide: WORKER_UI_STARTABLE_MESSAGING_SERVICE,
    useExisting: MessageBasedRenderer2, multi: true},
  BROWSER_SANITIZATION_PROVIDERS,
  {provide: ErrorHandler, useFactory: _exceptionHandler, deps: []},
  {provide: DOCUMENT, useFactory: _document, deps: []},
  {
    provide: EVENT_MANAGER_PLUGINS,
    useClass: DomEventsPlugin,
    deps: [DOCUMENT, NgZone],
    multi: true
  },
  {provide: EVENT_MANAGER_PLUGINS, useClass: KeyEventsPlugin,
    deps: [DOCUMENT], multi: true},
  {
    provide: EVENT_MANAGER_PLUGINS,
    useClass: HammerGesturesPlugin,
    deps: [DOCUMENT, HAMMER_GESTURE_CONFIG],
    multi: true
  },
  {provide: HAMMER_GESTURE_CONFIG, useClass: HammerGestureConfig, deps: []},
  APP_ID_RANDOM_PROVIDER,
  {provide: DomRendererFactory2, deps: [EventManager, DomSharedStylesHost]},
  {provide: RendererFactory2, useExisting: DomRendererFactory2},
  {provide: SharedStylesHost, useExisting: DomSharedStylesHost},
  {
    provide: ServiceMessageBrokerFactory,
    useClass: ServiceMessageBrokerFactory,
    deps: [MessageBus, Serializer]
  }
]

```



```

    },
    {
      provide: ClientMessageBrokerFactory,
      useClass: ClientMessageBrokerFactory,
      deps: [MessageBus, Serializer]
    },
    {provide: Serializer, deps: [RenderStore]},
    {provide: ON_WEB_WORKER, useValue: false},
    {provide: RenderStore, deps: []},
    {provide: DomSharedStylesHost, deps: [DOCUMENT]},
    {provide: Testability, deps: [NgZone]},
    {provide: EventManager, deps: [EVENT_MANAGER_PLUGINS, NgZone]},
    {provide: WebWorkerInstance, deps: []},
    {
      provide: PLATFORM_INITIALIZER,
      useFactory: initWebWorkerRenderPlatform,
      multi: true,
      deps: [Injector]
    },
    {provide: PLATFORM_ID, useValue: PLATFORM_WORKER_UI_ID},
    {provide: MessageBus, useFactory: messageBusFactory, deps:
[WebWorkerInstance]},
  ];

```

Note the provider configuration for `WORKER_UI_STARTABLE_MESSAGING_SERVICE` is set to `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 web worker and its message bus (note the `init` method just initializes the two fields):

```

/**
 * Wrapper class that exposes the Worker
 * and underlying {@link MessageBus} for lower level message passing.
 */
@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 () => {
    1 BrowserDomAdapter.makeCurrent();
    BrowserGetTestability.init();
    let scriptUri: string;
    try {
      2 scriptUri = injector.get(WORKER_SCRIPT);
    } catch (e) {
      throw new Error(
        'You must provide your WebWorker\'s

```

```

        initialization script with the WORKER_SCRIPT token');
    }
    const instance = injector.get(WebWorkerInstance);
    3 spawnWebWorker(scriptUri, instance);
    4 initializeGenericWorkerRenderer(injector);
  };
}

```

It returns a function that makes the browser Dom adapter 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:

```

// Spawns a new class and initializes the WebWorkerInstance
function spawnWebWorker(uri: string, instance: WebWorkerInstance): void {
  1 const webWorker: Worker = new Worker(uri);
  2 const sink = new PostMessageBusSink(webWorker);
  const source = new PostMessageBusSource(webWorker);
  const bus = new PostMessageBus(sink, source);
  3 instance.init(webWorker, bus);
}

```

It first creates a new web worker **1**; then it create the message bus with its sink and source **2** and finally it calls `WebWorkerInstance.init` **3** which we have already seen. The `initializeGenericWorkerRenderer` function is defined as:

```

function initializeGenericWorkerRenderer(injector: Injector) {
  const bus = injector.get(MessageBus);
  const zone = injector.get<NgZone>(NgZone);
  bus.attachToZone(zone);

  // initialize message services after the bus has been created
  const 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 three 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.

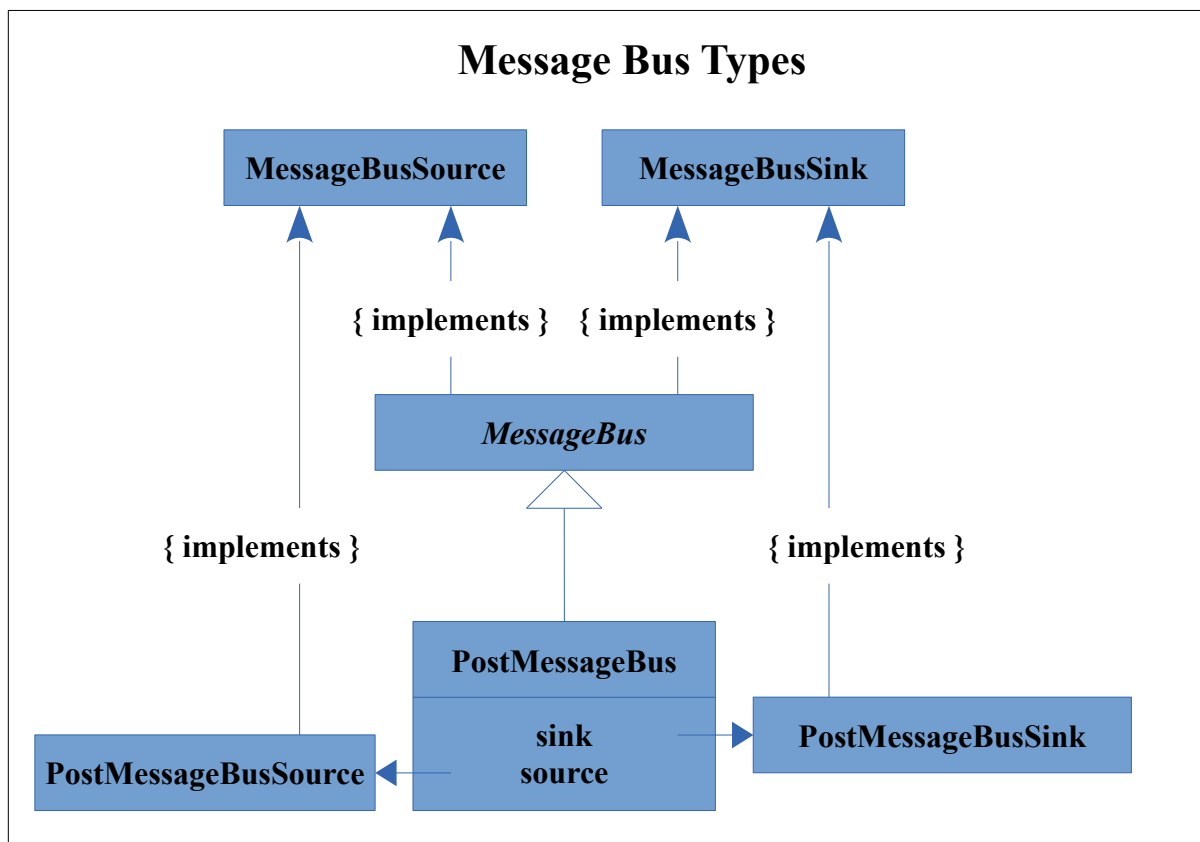
- `shared/api.ts`
- `shared/client_message_broker.ts`
- `shared/message_bus.ts`
- `shared/messaging_api.ts`
- `shared/post_message_bus.ts`
- `shared/render_store.ts`
- `shared/serializer.ts`
- `shared/service_message_broker.ts`
- `ui/event_dispatcher.ts`
- `ui/event_serializer.ts`

- ui/location_providers.ts
- ui/platform_location.ts
- ui/renderer.ts
- worker/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_2_CHANNEL = 'v2.ng-Renderer';
export const EVENT_2_CHANNEL = 'v2.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. Note the 2 in the names. This means these channels are for the updated renderer architecture.



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 attach to 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, attaching 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 TypeScript implementation of {@link MessageBus} for communicating
 * via JavaScript's postMessage API.
 */
@Injectable()
export class PostMessageBus implements MessageBus {...}

```

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){}
}

```

Channels are initialized with `PostMessageBusSource.initChannel()`:

```

initChannel(channel: string, runInZone: boolean = true) {
  if (this._channels.hasOwnProperty(channel)) {
    throw new Error(`${channel} has already been initialized`);
  }
  initChannel(channel: string, runInZone: boolean = true) {
    if (this._channels.hasOwnProperty(channel)) {
      throw new Error(`${channel} has already been initialized`);
    }

    const emitter = new EventEmitter(false);
    const channelInfo = new _Channel(emitter, runInZone);
    this._channels[channel] = channelInfo;
  }
}

```

So we see a channel is nothing more than 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 (this._channels.hasOwnProperty(channel)) {
    return this._channels[channel].emitter;
  } else {
    throw new Error(`${channel} is not set up. Did you forget to call
    initChannel?`);
  }
}

```

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` function passes on each message to the `_handleMessage` function, which emits it on the appropriate channel:

```

private _handleMessage(data: any): void {

```

```

const channel = data.channel;
if (this._channels.hasOwnProperty(channel)) {
  const 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} = {};
  private _messageBuffer: Array<Object> = [];

  constructor(private _postMessageTarget: PostMessageTarget) {}

```

The `initChannel` method subscribes to the emitter with a next handler that either (when running inside the Angular zone) adds the message to the `messageBuffer` where its sending is deferred, or (if running outside the Angular zone), calls `_sendMessages`, to immediately send the message:

```

initChannel(channel: string, runInZone: boolean = true): void {
  if (this._channels.hasOwnProperty(channel)) {
    throw new Error(`${channel} has already been initialized`);
  }

  const emitter = new EventEmitter(false);
  const channelInfo = new _Channel(emitter, runInZone);
  this._channels[channel] = channelInfo;
  emitter.subscribe((data: Object) => {
    const 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

- [<ANGULAR-MASTER>/packages/core/src/zone/ngzone.ts](https://github.com/angular/angular/blob/master/packages/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.
 */
readonly onStable: EventEmitter<any> = new EventEmitter(false);

```

So when the zone has no more work to immediately 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:

```

/**
 * A TypeScript implementation of {@link MessageBus} for communicating
 * via JavaScript's postMessage API.
 */
@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); }

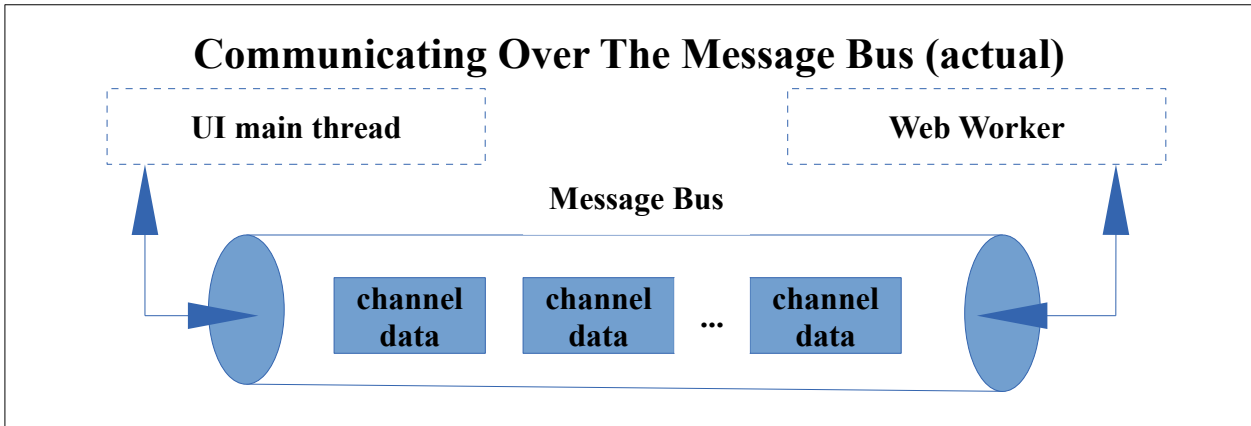
    to(channel: string): EventEmitter<any> { return this.sink.to(channel); }
}

/**
 * Helper class that wraps a channel's {@link EventEmitter} and
 * keeps track of if it should run in the zone.
 */

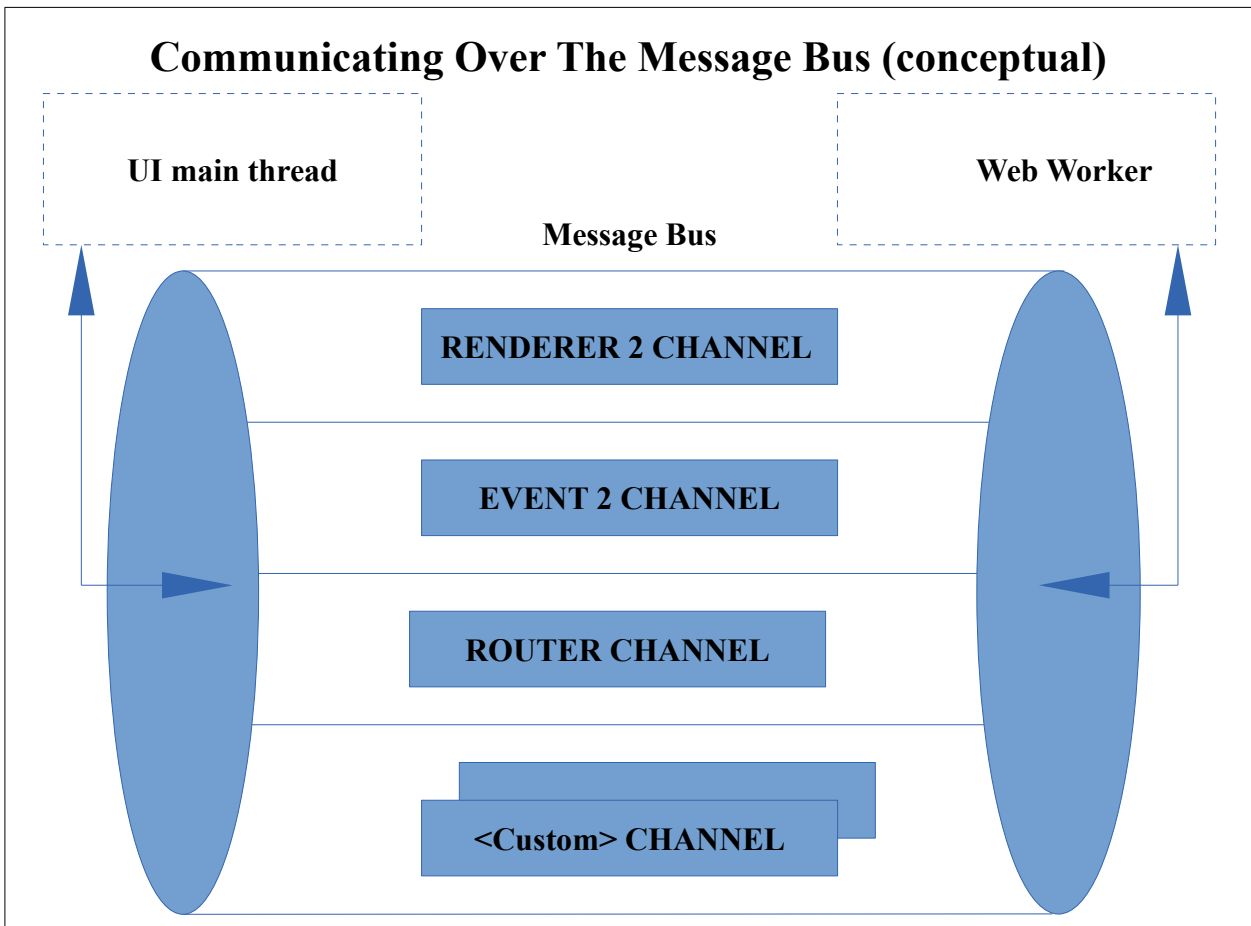
```



```
class _Channel {  
  constructor(public emitter: EventEmitter<any>, public runInZone: boolean) {}  
}
```



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 web workers. It is based on a single connection, with opaque messages consisting of a channel name (string) and message data:

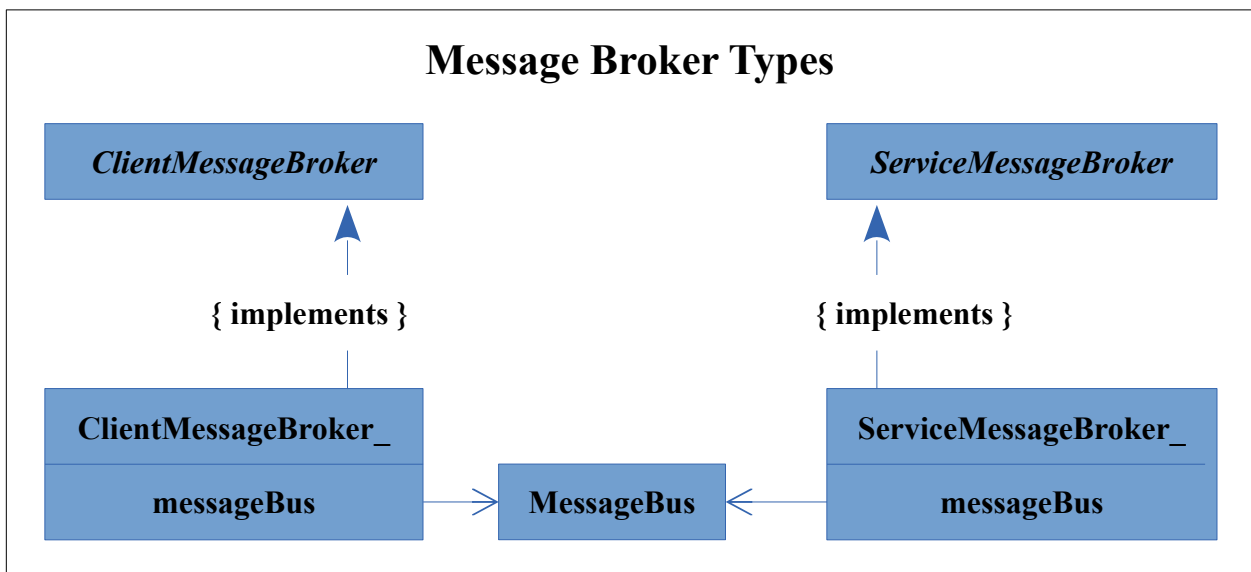
```
var msg = {channel: channel, message: data};
```

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 interface ReceivedMessage {
  method: string;
  args: any[];
  id: string;
  type: string;
}
```



The class `ServiceMessageBrokerFactory` provide a factory for the service message broker:

```
@Injectable()
export class ServiceMessageBrokerFactory {
  /** @internal */
  _serializer: Serializer;

  /** @internal */
  constructor(private _messageBus: MessageBus, _serializer: Serializer) {
    this._serializer = _serializer;
  }

  /**
   * Initializes the given channel and attaches a new {@link
   ServiceMessageBroker} to it.
   */
  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:

```
/**
 * Helper class for UIComponents that allows components to register methods.
 * If a registered method message is received from the broker on the worker,
 * the UIMessageBroker deserializes its arguments and calls the
 * registered method. If that method returns a promise, the UIMessageBroker
 * returns the result to the worker.
 */
export class ServiceMessageBroker {
  private _sink: EventEmitter<any>;
  private _methods = new Map<string, Function>();

  /** @internal */
  constructor(messageBus: MessageBus,
    private _serializer: Serializer, private channel: string) {
    this._sink = messageBus.to(channel);
    const 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(message: ReceivedMessage): void {
  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. Its key is the `methodName` supplied as a parameter and its value is an anonymous method, with a single parameter, `message`, of type `ReceivedMessage`:

```
registerMethod(
  methodName: string, signature: Array<Type<any>|SerializerTypes>|null,
  method: (..._: any[]) => Promise<any>| void,
  returnType?: Type<any>|SerializerTypes): void {
  this._methods.set(methodName, (message: ReceivedMessage) => {
    const serializedArgs = message.args;
    const numArgs = signature ? signature.length : 0;
    const deserializedArgs = new Array(numArgs);
    for (let i = 0; i < numArgs; i++) {
      const serializedArg = serializedArgs[i];
      1 deserializedArgs[i] =
        this._serializer.deserialize(serializedArg, signature![i]);
    }
  })
}
```

```

2   const promise = method(...deserializedArgs);
      if (returnType && promise) {
3     this._wrapWebWorkerPromise(message.id, promise, returnType);
      }
    });
  }

```

We see at its **1** deserialization occurring 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>|SerializerTypes): 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;
}
interface RequestMessageData {
  method: string;
  args?: any[];
  id?: string;
}
interface ResponseMessageData {
  type: 'result'|'error';
  value?: any;
  id?: string;
}
export class FnArg {
  constructor(
    public value: any, public type: Type<any>|SerializerTypes =
    SerializerTypes.PRIMITIVE) {}
}
export class UiArguments {
  constructor(public method: string, public args?: FnArg[]) {}
}

```

To construct a client message broker it defines the `ClientMessageBrokerFactory` which, in its `createMessageBroker` method, calls `initChannel` on the message bus and returns a new `ClientMessageBroker`:

```

@Injectable()
export class ClientMessageBrokerFactory {
  _serializer: Serializer;
  constructor(private _messageBus: MessageBus, _serializer: Serializer) {

```

```

    this._serializer = _serializer;
  }

  // Initializes the given channel and attaches a new
  // {@link 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 an important method to 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). `ClientMessageBroker` is implemented as follows:

```

export class ClientMessageBroker {
  private _pending = new Map<string, PromiseCompleter>();
  private _sink: EventEmitter<any>;
  public _serializer: Serializer;

  constructor(
    messageBus: MessageBus, _serializer: Serializer, private channel: any) {
    this._sink = messageBus.to(channel);
    this._serializer = _serializer;
    const source = messageBus.from(channel);

    source.subscribe(
      {next: (message: ResponseMessageData) => 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 {
  const time: string = stringify(new Date().getTime());
  let iteration: number = 0;
  let id: string = name + time + stringify(iteration);
  while (this._pending.has(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>|SerializerTypes|null): Promise<any>|null {
  const fnArgs: any[] = [];
  if (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);
      }
    });
  }

  let promise: Promise<any>|null;
  let id: string|null = null;

  if (returnType !== null) { .. }
2  const message: RequestMessageData = {
    'method': args.method,
    'args': fnArgs,
  };
  if (id !== null) {
    message['id'] = id;
  }
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 = undefined !;
  promise =
    new Promise((resolve, reject) => { completer = {resolve, reject}; });
2  id = this._generateMessageId(args.method);
3  this._pending.set(id, completer);

  promise.catch((err) => {
    if (console && console.error) {
      // tslint:disable-next-line:no-console
      console.error(err);
    }

    completer.reject(err);
  });

4  promise = promise.then(
```

```

5a         (v: any) => this._serializer ?
5b         this._serializer.deserialize(v, returnType) : v);
    } else {
      promise = null;
    }
  }
  const message: RequestMessageData = {
    'method': args.method,
    'args': fnArgs,
  };
  if (id !== null) {
6    message['id'] = id;
  }
  this._sink.emit(message);

```

The `_handleMessage` accepts a `ResponseMessageData` parameter. It extracts **1** 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: ResponseMessageData): void {
  if (message.type === 'result' || message.type === 'error') {
1    const id = message.id !;
2    if (this._pending.has(id)) {
      if (message.type === 'result') {
3        this._pending.get(id) !.resolve(message.value);
      } else {
4        this._pending.get(id) !.reject(message.value);
      }
    }
    this._pending.delete(id);
  }
}

```

Two helper files are involved with serialization – `render_store.ts` and `serializer.ts`. The `RenderStore` class, define in `render_store.ts`, manages 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 `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: Type<any>|SerializerTypes =
    SerializerTypes.PRIMITIVE): Object {
    if (obj == null || type === SerializerTypes.PRIMITIVE) {
      return obj;
    }
    if (Array.isArray(obj)) {
      return obj.map(v => this.serialize(v, type));
    }
    if (type === SerializerTypes.RENDER_STORE_OBJECT) {
      return this._renderStore.serialize(obj) !;
    }
    if (type === RenderComponentType) {

```

```

    return this._serializeRenderComponentType(obj);
  }
  if (type === SerializerTypes.RENDERER_TYPE_2) {
    return this._serializeRenderType2(obj);
  }
  if (type === LocationType) {
    return this._serializeLocation(obj);
  }
  throw new Error(`No serializer for type ${stringify(type)}`);
}

deserialize(map: any, type: Type<any>|SerializerTypes =
SerializerTypes.PRIMITIVE, data?: any):
  any {
  if (map == null || type === SerializerTypes.PRIMITIVE) {
    return map;
  }
  if (Array.isArray(map)) {
    return map.map(val => this.deserialize(val, type, data));
  }
  if (type === SerializerTypes.RENDER_STORE_OBJECT) {
    return this._renderStore.deserialize(map);
  }
  if (type === RenderComponentType) {
    return this._deserializeRenderComponentType(map);
  }
  if (type === SerializerTypes.RENDERER_TYPE_2) {
    return this._deserializeRenderType2(map);
  }
  if (type === LocationType) {
    return this._deserializeLocation(map);
  }
  throw new Error(`No deserializer for type ${stringify(type)}`);
}

private _serializeLocation(loc: LocationType): Object {
  return {
    'href': loc.href,
    'protocol': loc.protocol,
    'host': loc.host,
    'hostname': loc.hostname,
    'port': loc.port,
    'pathname': loc.pathname,
    'search': loc.search,
    'hash': loc.hash,
    'origin': loc.origin,
  };
}

private _deserializeLocation(loc: {[key: string]: any}): LocationType {
  return new LocationType(
    loc['href'], loc['protocol'], loc['host'], loc['hostname'],
loc['port'], loc['pathname'],
    loc['search'], loc['hash'], loc['origin']);
}

private _serializeRenderComponentType(type: RenderComponentType): Object {
  return {
    'id': type.id,

```



```

        'templateUrl': type.templateUrl,
        'slotCount': type.slotCount,
        'encapsulation': this.serialize(type.encapsulation),
        'styles': this.serialize(type.styles),
    };
}

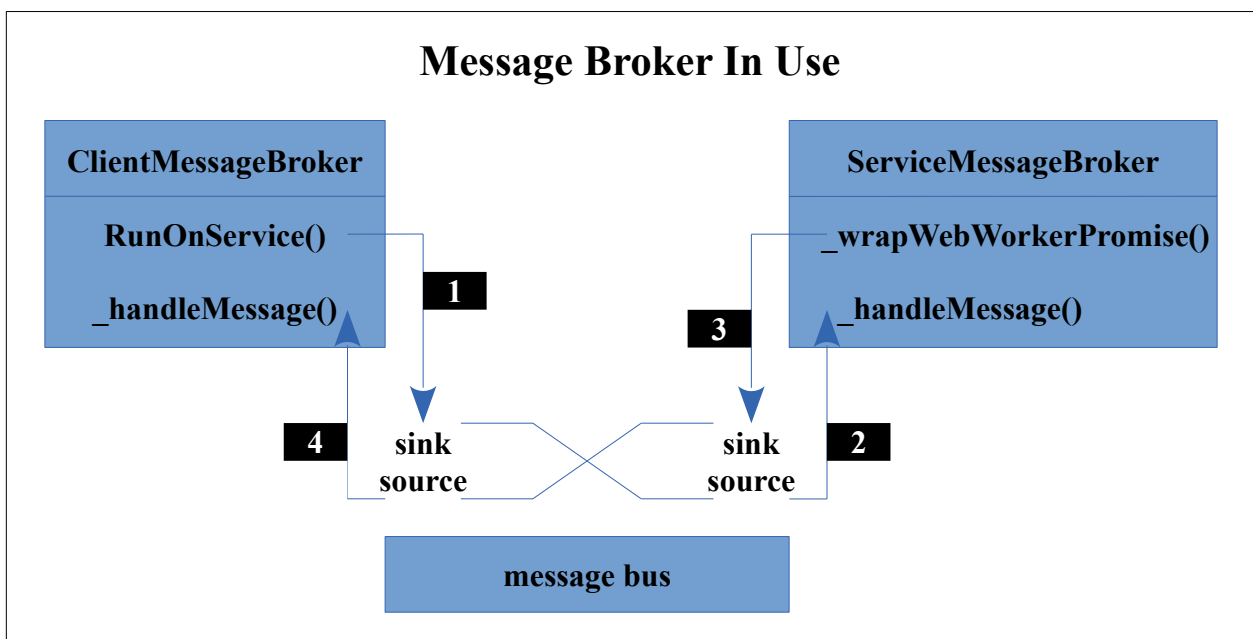
private _deserializeRenderComponentType(props: {[key: string]: any}):
RenderComponentType {
    return new RenderComponentType(
        props['id'], props['templateUrl'], props['slotCount'],
        this.deserialize(props['encapsulation']),
        this.deserialize(props['styles']), {});
}

private _serializeRendererType2(type: RendererType2): {[key: string]: any}
{
    return {
        'id': type.id,
        'encapsulation': this.serialize(type.encapsulation),
        'styles': this.serialize(type.styles),
        'data': this.serialize(type.data),
    };
}

private _deserializeRendererType2(props: {[key: string]: any}):
RendererType2 {
    return {
        id: props['id'],
        encapsulation: props['encapsulation'],
        styles: this.deserialize(props['styles']),
        data: this.deserialize(props['data'])
    };
}
}

```

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 InjectionToken<boolean>('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:

- [<ANGULAR-MASTER>/packages/platform-webworker/src/worker_app.ts](#)

has this:

```
// The ng module for the worker app side.
@NgModule({
  providers: [
    ..
    {provide: ON_WEB_WORKER, useValue: true},
    RenderStore,
    ..
  ],
  exports: [
    CommonModule,
    ApplicationModule,
  ]
})
export class WorkerAppModule { }
```

Note `ON_WEB_WORKER` is set to true.

On the ui main thread side:

- [<ANGULAR-MASTER>/packages/platform-webworker/src/worker_render.ts](#)

has this:

```
export const _WORKER_UI_PLATFORM_PROVIDERS: StaticProvider[] = [
  ..
  {provide: Serializer, deps: [RenderStore]},
  {provide: ON_WEB_WORKER, useValue: false},
  {provide: RenderStore, deps: []},
  ..
];

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:

- [<ANGULAR-MASTER>/packages/platform-webworker/src/web_workers](#)

In `worker_adapter.ts`, `WorkerDomAdapter` extends `DomAdapter` 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.

```

/**
 * This adapter is required to log error messages.
 *
 * Note: other methods all throw as the DOM is not accessible
 * directly in web worker context.
 */
export class WorkerDomAdapter extends DomAdapter {

    static makeCurrent() { setRootDomAdapter(new WorkerDomAdapter()); }

    log(error: any) {
        // tslint:disable-next-line:no-console
        console.log(error);
    }

    // everything erlse throws an exception
    insertAllBefore(parent: any, el: any, nodes: any) {
        throw 'not implemented'; }
    insertAfter(parent: any, el: any, node: any) { throw 'not implemented'; }
    setInnerHTML(el: any, value: any) { throw 'not implemented'; }
}

```

The `renderer.ts` file supplies the `WebWorkerRendererFactory2` and `WebWorkerRenderer2` classes and this is where worker-based rendering is managed.

`WebWorkerRenderer2` implements the `Renderer` API and forwards all calls to `runOnService` from the root renderer. `WebWorkerRenderer2` is running the webworker 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 WebWorkerRenderer2 implements Renderer2 {
    data: {[key: string]: any} = Object.create(null);

    constructor(private _rendererFactory: WebWorkerRendererFactory2) {}

    private asFnArg = new FnArg(this, SerializerTypes.RENDER_STORE_OBJECT);

    destroy(): void { this.callUIWithRenderer('destroy'); }

    createElement(name: string, namespace?: string): any {
        const node = this._rendererFactory.allocateNode();
        this.callUIWithRenderer('createElement', [
            new FnArg(name),
            new FnArg(namespace),
            new FnArg(node, SerializerTypes.RENDER_STORE_OBJECT),
        ]);
        return node;
    }

    createComment(value: string): any {
        const node = this._rendererFactory.allocateNode();
        this.callUIWithRenderer('createComment', [
            new FnArg(value),
            new FnArg(node, SerializerTypes.RENDER_STORE_OBJECT),
        ]);
        return node;
    }
}

```

```

    }

    appendChild(parent: any, newChild: any): void {
      this.callUIWithRenderer('appendChild', [
        new FnArg(parent, SerializerTypes.RENDER_STORE_OBJECT),
        new FnArg(newChild, SerializerTypes.RENDER_STORE_OBJECT),
      ]);
    }
  }

```

WebWorkerRendererFactory2 implements Core's `RendererFactory2` by setting up the client message broker factory.

```

@Injectables()
export class WebWorkerRendererFactory2 implements RendererFactory2 {
  globalEvents = new NamedEventEmitter();

  private _messageBroker: ClientMessageBroker;

  constructor(
    messageBrokerFactory: ClientMessageBrokerFactory, bus: MessageBus,
    private _serializer: Serializer, public renderStore: RenderStore) {
    this._messageBroker =
      messageBrokerFactory.createMessageBroker(RENDERER_2_CHANNEL);
    1 bus.initChannel(EVENT_2_CHANNEL);
    2 const source = bus.from(EVENT_2_CHANNEL);
    3 source.subscribe({next: (message: any) => this._dispatchEvent(message)});
  }
}

```

An additional and very important role of `WebWorkerRendererFactory2` is to configure event handling. We see at **1** `initChannel()` being called for the `EVENT_2_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 {
  const element: WebWorkerRenderNode =
    this._serializer.deserialize(
      message['element'], SerializerTypes.RENDER_STORE_OBJECT);

  const eventName = message['eventName'];
  const target = message['eventTarget'];
  const event = message['event'];

  if (target) {
    this.globalEvents.dispatchEvent(eventNameWithTarget(target, eventName),
      event);
  } else {
    element.events.dispatchEvent(eventName, event);
  }
}

```

The ui specific code is in:

- [/packages/platform-webworker/src/web_workers/ui](https://github.com/angular/angular/blob/master/packages/platform-webworker/src/web_workers/ui)

The `renderer.ts` file implements the `MessageBasedRenderer2` class. Note this class, despite its name, does not implement any types from Core's rendering API. Instead, it accepts a `RendererFactory2` 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 MessageBasedRenderer2 {
  private _eventDispatcher: EventDispatcher;

  constructor(
    private _brokerFactory: ServiceMessageBrokerFactory,
    private _bus: MessageBus,
    private _serializer: Serializer, private _renderStore: RenderStore,
    private _rendererFactory: RendererFactory2) {}

```

Its `start()` method initializes the `EVENT_2_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 {
  const broker =
    this._brokerFactory.createMessageBroker(RENDERER_2_CHANNEL);

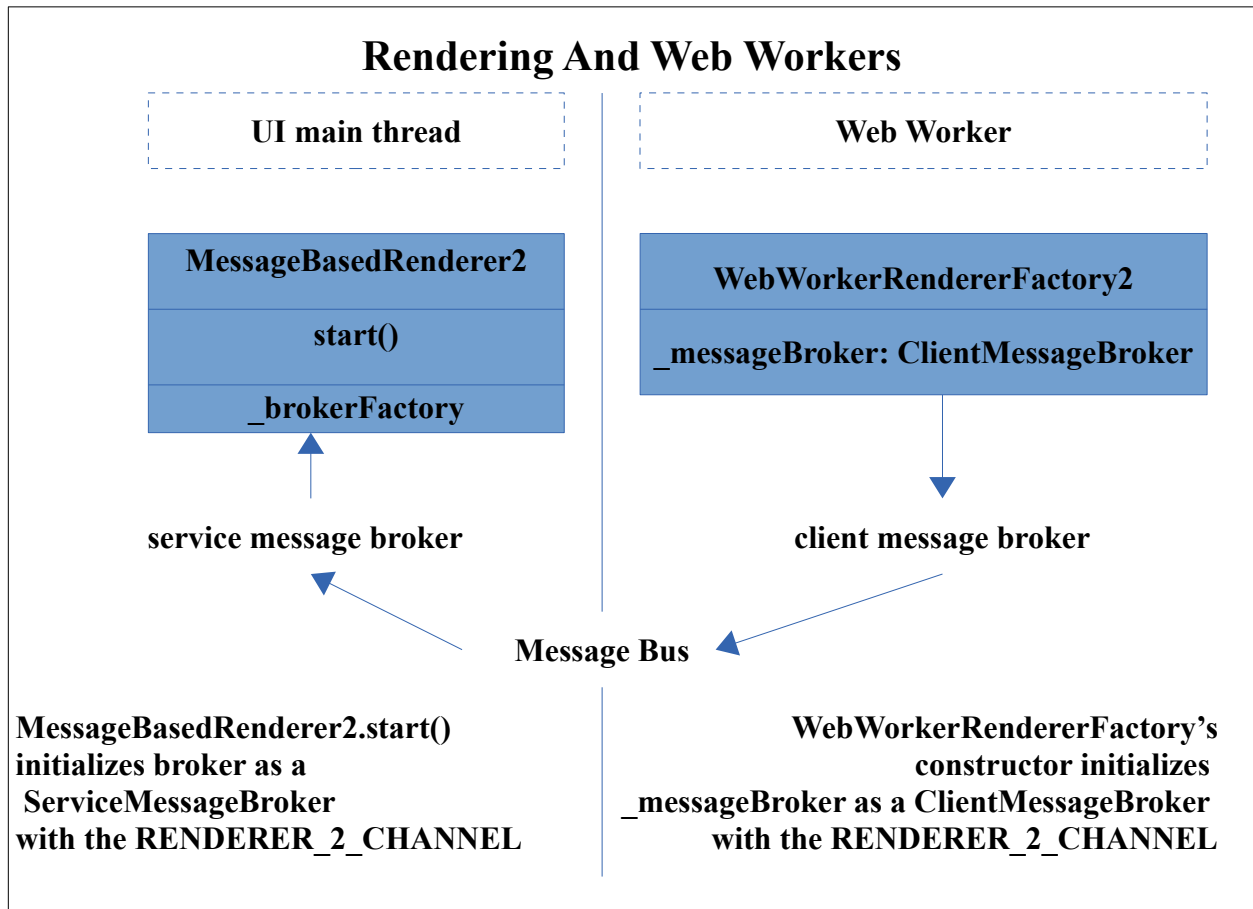
  this._bus.initChannel(EVENT_2_CHANNEL);
  this._eventDispatcher =
    new EventDispatcher(this._bus.to(EVENT_2_CHANNEL), this._serializer);

  const [RSO, P, CRT] = [
    SerializerTypes.RENDER_STORE_OBJECT,
    SerializerTypes.PRIMITIVE,
    SerializerTypes.RENDERER_TYPE_2,
  ];

  const methods: any[][] = [
    ['createRenderer', this.createRenderer, RSO, CRT, P],
    ['createElement', this.createElement, RSO, P, P, P],
    ['createComment', this.createComment, RSO, P, P],
    ['createText', this.createText, RSO, P, P],
    ['appendChild', this.appendChild, RSO, RSO, RSO],
    ['insertBefore', this.insertBefore, RSO, RSO, RSO, RSO],
    ['removeChild', this.removeChild, RSO, RSO, RSO],
    ['selectRootElement', this.selectRootElement, RSO, P, P],
    ['parentNode', this.parentNode, RSO, RSO, P],
    ['nextSibling', this.nextSibling, RSO, RSO, P],
    ['setAttribute', this.setAttribute, RSO, RSO, P, P, P],
    ['removeAttribute', this.removeAttribute, RSO, RSO, P, P],
    ['addClass', this.addClass, RSO, RSO, P],
    ['removeClass', this.removeClass, RSO, RSO, P],
    ['setStyle', this.setStyle, RSO, RSO, P, P, P],
    ['removeStyle', this.removeStyle, RSO, RSO, P, P],
    ['setProperty', this.setProperty, RSO, RSO, P, P],
    ['setValue', this.setValue, RSO, RSO, P],
    ['listen', this.listen, RSO, RSO, P, P, P],
    ['unlisten', this.unlisten, RSO, RSO],
    ['destroy', this.destroy, RSO],
    ['destroyNode', this.destroyNode, RSO, P]
  ];

  methods.forEach(([name, method, ...argTypes]: any[]) => {
    broker.registerMethod(name, argTypes, method.bind(this));
  });
}

```



The local methods usually call the equivalent method in the configured renderer, and often stores the result in the `RenderStore`. Here is `_createElement()`:

```
private createElement(r: Renderer2, name: string,
                    namespace: string, id: number) {
  this._renderStore.store(r.createElement(name, namespace), id);
}
```

For event listening, the event dispatcher is used:

```
private listen(r: Renderer2, el: any, elName: string,
             eventName: string, unlistenId: number) {
  const listener = (event: any) => {
    return this._eventDispatcher.dispatchRenderEvent(
      el, elName, eventName, event);
  };

  const unlisten = r.listen(el || elName, eventName, listener);
  this._renderStore.store(unlisten, unlistenId);
}
```

The `EventDispatcher` class is defined in `event_dispatcher.ts` and has a constructor and one method, `dispatchRenderEvent()`:

```
export class EventDispatcher {
  constructor(private _sink: EventEmitter<any>,
             private _serializer: Serializer) {}
}
```

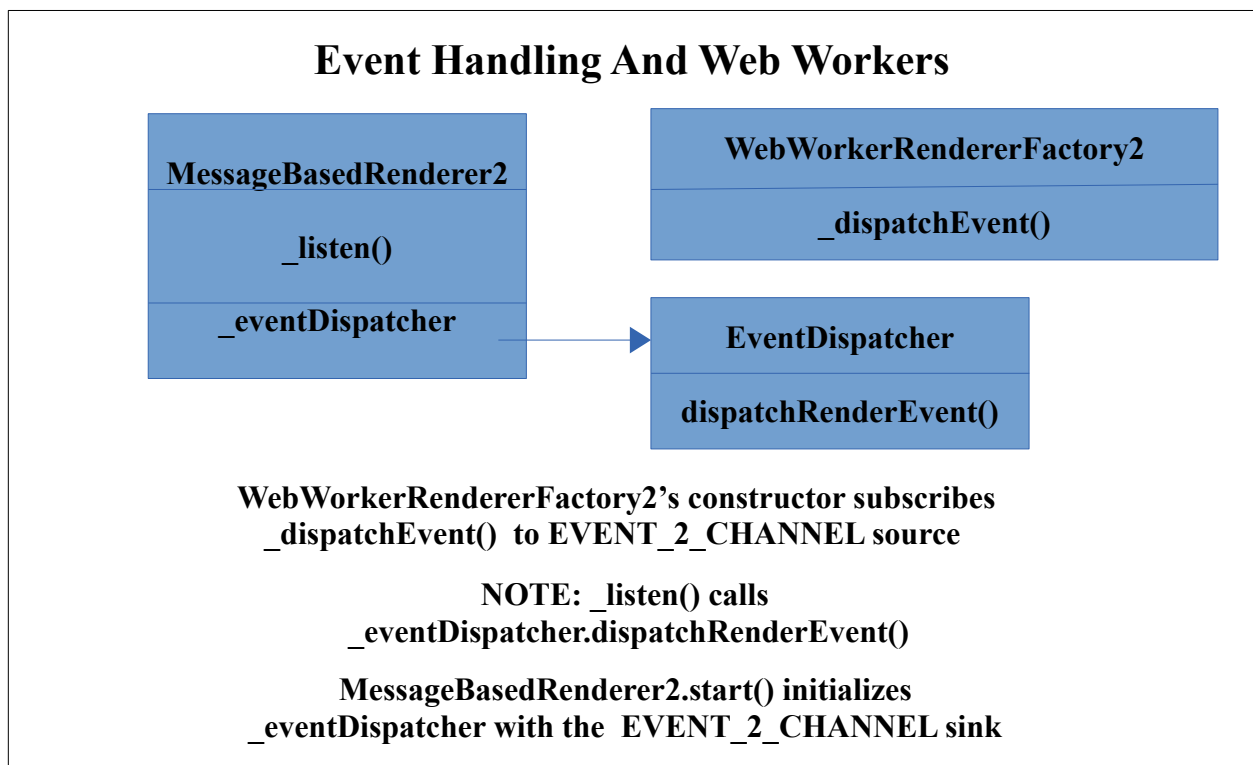
```

dispatchRenderEvent(element: any, eventTarget: string,
  eventName: string, event: any): boolean {
  let serializedEvent: any;

  switch (event.type) {
    ..
    case 'keydown':
    case 'keypress':
    case 'keyup':
      serializedEvent = serializeKeyboardEvent(event);
      break;
    ..
    default:
      throw new Error(eventName + ' not supported on WebWorkers');
  }
  this._sink.emit({
    'element': this._serializer.serialize(
      element, SerializerTypes.RENDER_STORE_OBJECT),
    '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 it has for keyboard events.



12: The Platform-WebWorker-Dynamic Package

Overview

Platform-WebWorkers-Dynamic is the smallest of all the Angular packages. It defines 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 is 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 Platform-WebWorker-Dynamic package can be represented as:

@Angular/Platform-WebWorker-Dynamic API

platformWorkerAppDynamic

Source Tree Layout

The source tree for the Platform-WebWorker-Dynamic package contains this directory:

- `src`

Currently there are no test nor testing sub-directories. It also has these files:

- `index.ts`
- `package.json`
- `public_api.ts`
- `rollup.config.js`
- `tsconfig-build.json`

The `index.ts` file is simply:

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

The `public_api.ts` file is:


```
/**
 * @module
 * @description
 * Entry point for all public APIs of this package.
 */
export * from './src/platform-webworker-dynamic';
// This file only reexports content of the `src` folder. Keep it that way.
```

The `src/platform-webworker-dynamic.ts` file is:

```
// other imports
import {
  eResourceLoaderImpl as ResourceLoaderImpl,
  eplatformCoreDynamic as platformCoreDynamic}
from '@angular/platform-browser-dynamic';

createPlatformFactory(platformCoreDynamic, 'workerAppDynamic', [
  {
    provide: COMPILER_OPTIONS,
    useValue: {providers: [{provide: ResourceLoader,
      useClass: ResourceLoaderImpl, deps: []}]}, multi: true },
  {provide: PLATFORM_ID, useValue: PLATFORM_WORKER_UI_ID}
]);
```

Note that both `ResourceLoaderImpl` and `platformCoreDynamic` are imported from the `platform-browser-dynamic` package.

13: The Router Package

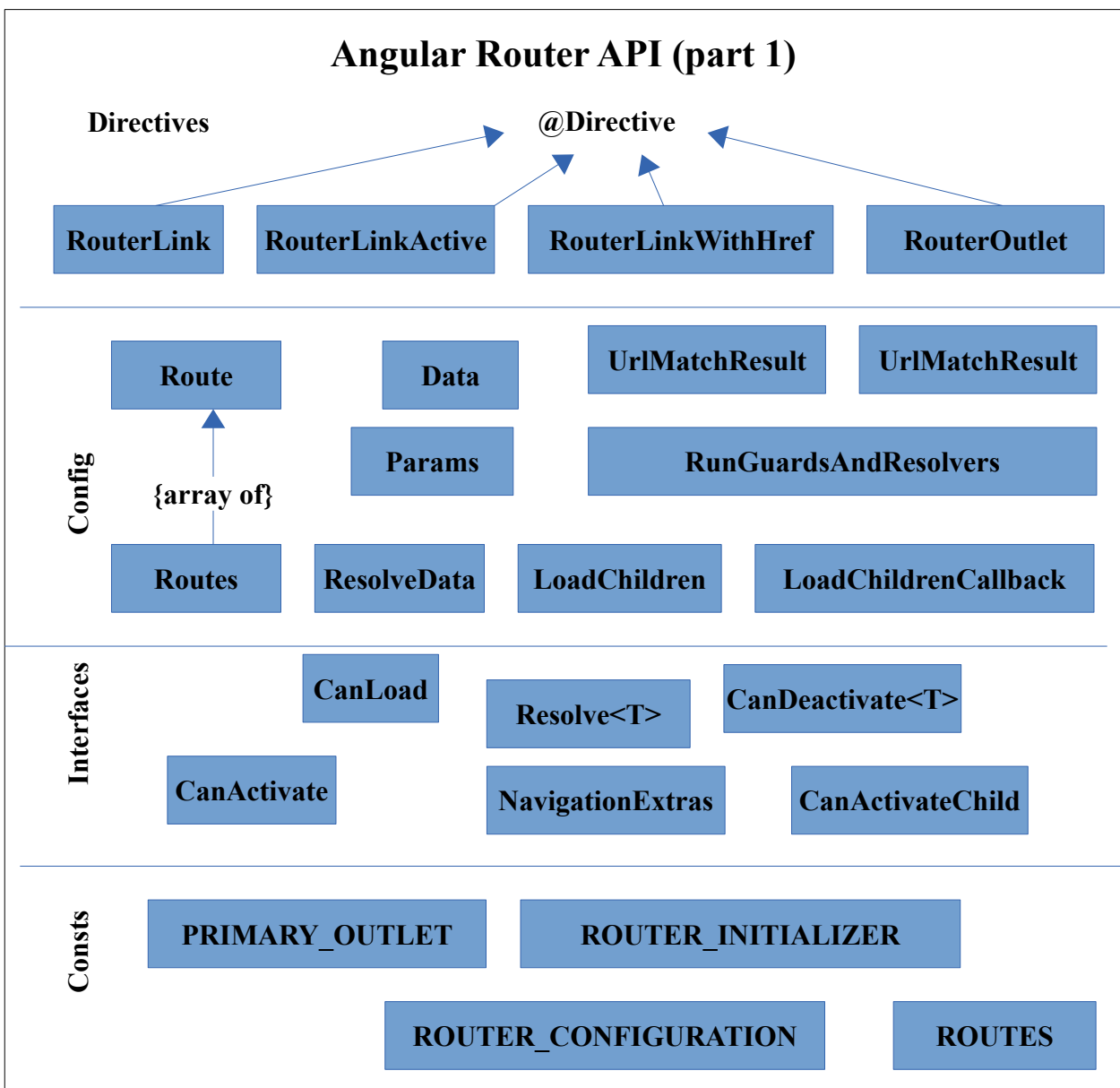
Overview

The Angular Router package provides functionality to:

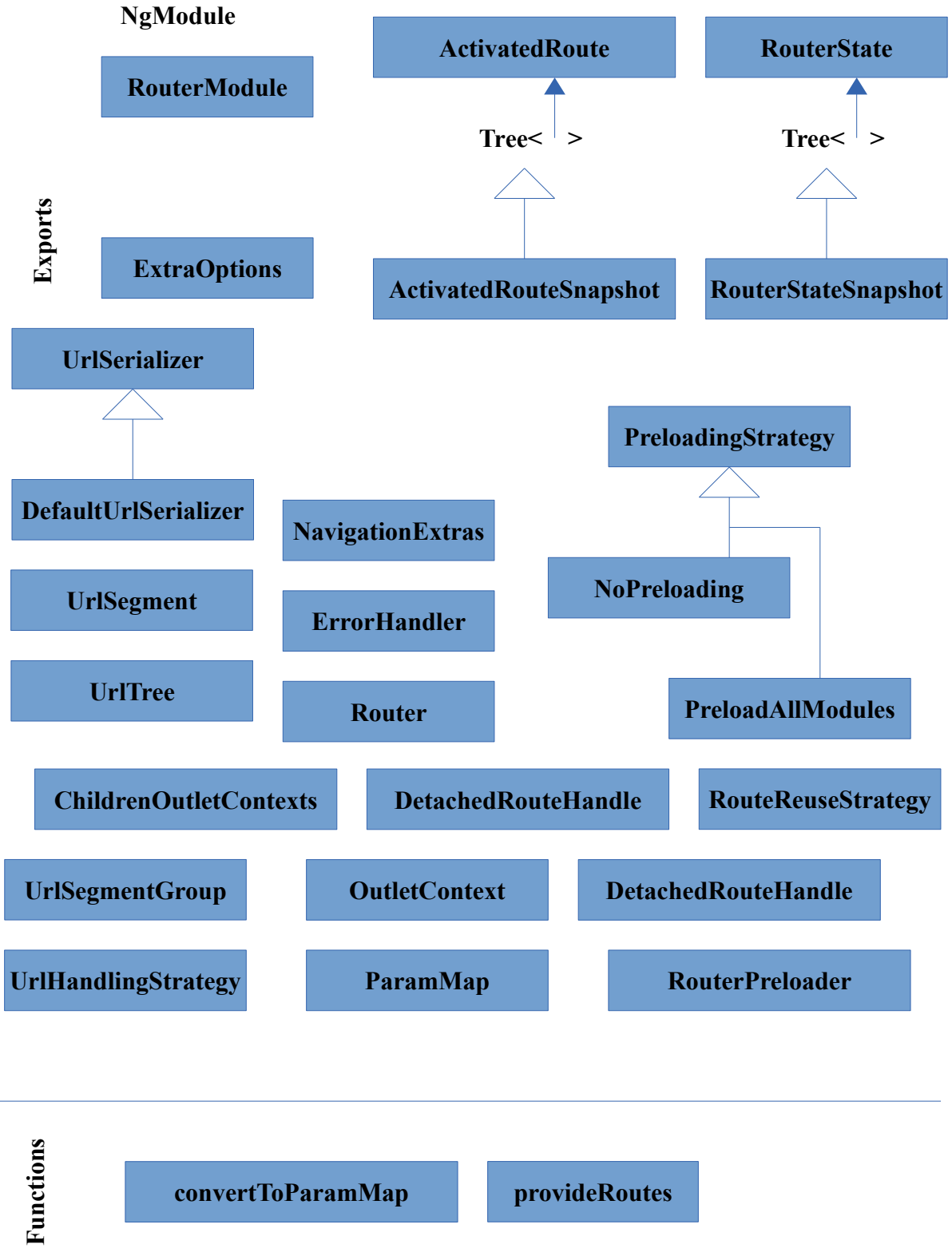
- manage navigation based on URLs and reflect state in the URL
- manage application state and state transitions
- Eagerly load, lazily-load and preload modules as needed

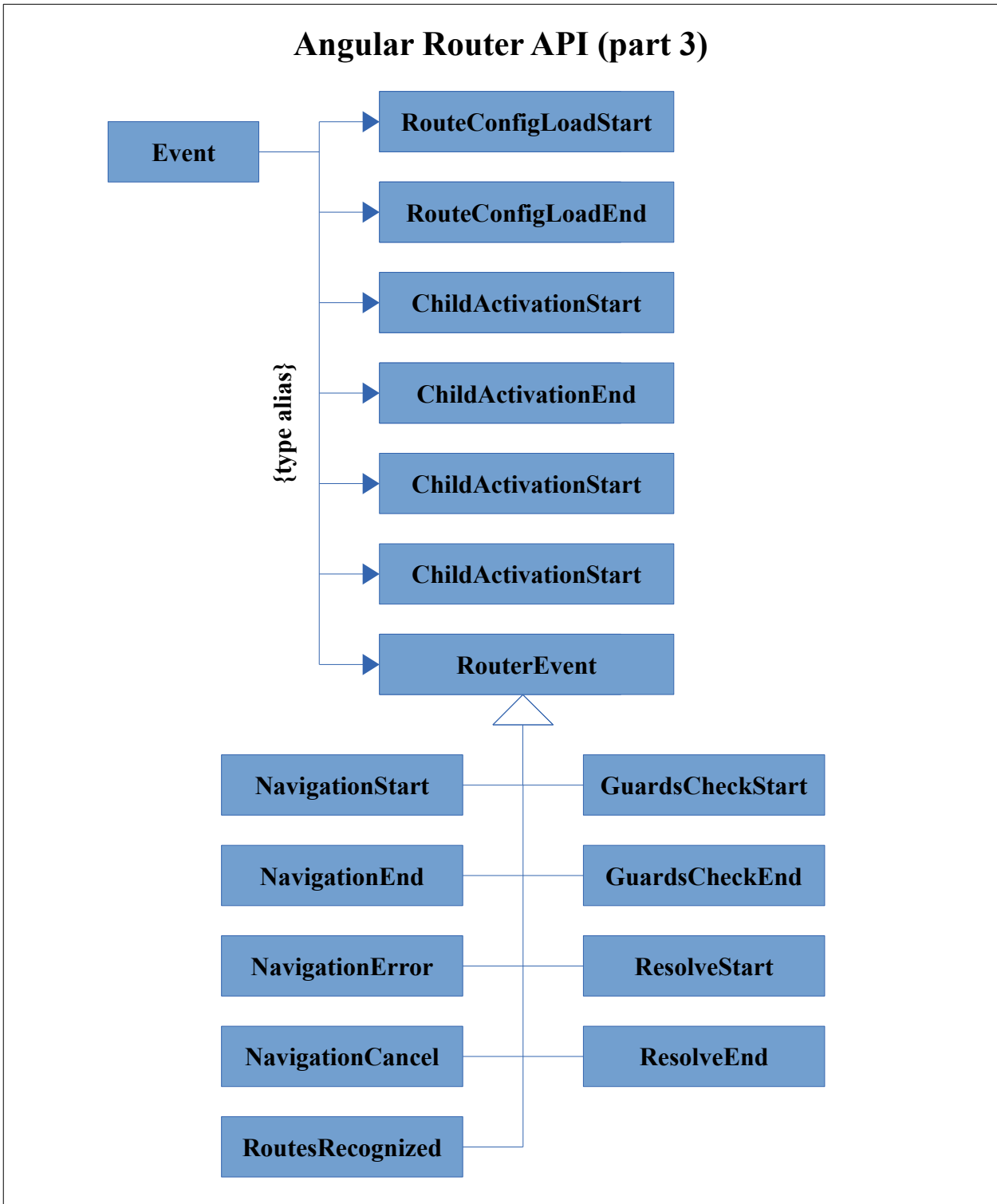
Router API

The exported API of the Angular Router package can be represented as:



Angular Router API (part 2)





The Angular Router source tree is at:

- [<ANGULAR-MASTER>/packages/router](https://github.com/angular/angular/tree/master/packages/router)

Its root directory contains index.ts, which just exports the contents of public_api.ts, which in turn exports the contents of .src/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,
  RunGuardsAndResolvers, UrlMatchResult, UrlMatcher} from './config';
export {RouterLink, RouterLinkWithHref} from './directives/router_link';
export {RouterLinkActive} from './directives/router_link_active';
export {RouterOutlet} from './directives/router_outlet';
export {ActivationEnd, ActivationStart, ChildActivationEnd,
  ChildActivationStart, Event, GuardsCheckEnd, GuardsCheckStart,
  NavigationCancel, NavigationEnd, NavigationError, NavigationStart,
  ResolveEnd, ResolveStart, RouteConfigLoadEnd, RouteConfigLoadStart,
  RouterEvent, RoutesRecognized} from './events';
export {CanActivate, CanActivateChild, CanDeactivate, CanLoad, Resolve}
  from './interfaces';
export {DetachedRouteHandle, RouteReuseStrategy}
  from './route_reuse_strategy';
export {NavigationExtras, Router} from './router';
export {ROUTES} from './router_config_loader';
export {ExtraOptions, ROUTER_CONFIGURATION, ROUTER_INITIALIZER, RouterModule,
  provideRoutes} from './router_module';
export {ChildrenOutletContexts, OutletContext}
  from './router_outlet_context';
export {NoPreloading, PreloadAllModules, PreloadingStrategy, RouterPreloader}
  from './router_preloader';
export {ActivatedRoute, ActivatedRouteSnapshot, RouterState,
  RouterStateSnapshot} from './router_state';
export {PRIMARY_OUTLET, ParamMap, Params, convertToParamMap} from './shared';
export {UrlHandlingStrategy} from './url_handling_strategy';
export {DefaultUrlSerializer, UrlSegment, UrlSegmentGroup, UrlSerializer,
  UrlTree} from './url_tree';
export {VERSION} from './version';
```

It also has the line:

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

As already discussed, such private exports are intended for other packages within the Angular Framework itself, and not to be directly used by Angular applications. The `private_export.ts` file has these exports (note names are prepended with `ɵ`);

```
export {ROUTER_PROVIDERS as ɵROUTER_PROVIDERS} from './router_module';
export {flatten as ɵflatten} from './utils/collection';
```

Source Tree Layout

The source tree for the Router package contains these directories:

- scripts
- src
- test (unit tests in Jasmine)
- testing (testing tools)

The Router package root directory contains these files:

- BUILD.bazel
- index.ts
- karma-test-shim.ts
- karma.config.js
- LICENSE
- package.json

- public_api.ts
- README.md
- rollup.config.js
- tsconfig-build.json

Source

router/src

The router/src directory contains:

- apply_redirects.ts
- config.ts
- create_router_state.ts
- create_url_tree.ts
- events.ts
- interfaces.ts
- index.ts
- interfaces.ts
- pre_activation.ts
- private_export.ts
- recognize.ts
- resolve.ts
- route_reuse_strategy.ts
- router_config_loader.ts
- router_module.ts
- router_outlet_context.ts
- router_preloader.ts
- router_state.ts
- router.ts
- shared.ts
- url_handling_strategy.ts
- url_tree.ts
- version.ts

We'll start by looking at:

- [<ANGULAR-MASTER>/packages/router/src/router_module.ts](#)

which defines the RouterModule class and related types.

It defines three consts:

```
const ROUTER_DIRECTIVES =  
  [RouterOutlet, RouterLink, RouterLinkWithHref, RouterLinkActive];
```

The first, `ROUTER_DIRECTIVES`, is the collection of router directives that can appear in Angular templates defining where the routed content is to be located on the page, and how links used for routing are to be displayed. `ROUTER_DIRECTIVES` is specified in the declarations and exports of the `@NgModule` metadata for `RouterModule`:

```
@NgModule({declarations: ROUTER_DIRECTIVES, exports: ROUTER_DIRECTIVES})  
export class RouterModule { .. }
```

The other two are injection tokens for DI:

```
export const ROUTER_CONFIGURATION =
  new InjectionToken<ExtraOptions>('ROUTER_CONFIGURATION');
export const ROUTER_FORROOT_GUARD =
  new InjectionToken<void>('ROUTER_FORROOT_GUARD');
```

It also defines the `ROUTER_PROVIDERS` array of providers (which is only used by `forRoot`, not `forChild`):

```
export const ROUTER_PROVIDERS: Provider[] = [
  Location,
  {provide: UrlSerializer, useClass: DefaultUrlSerializer},
  {provide: Router, useFactory: setupRouter, .. },
  ChildrenOutletContexts,
  {provide: ActivatedRoute, useFactory: rootRoute, deps: [Router]},
  {provide: NgModuleFactoryLoader, useClass: SystemJsNgModuleLoader},
  RouterPreloader,
  NoPreloading,
  PreloadAllModules,
  {provide: ROUTER_CONFIGURATION, useValue: {enableTracing: false}},
];
```

An important provider there is `Router`, which is the actually routing service. This is set up in DI to return the result of the `setupRouter` factory method. An abbreviated version of this is as follows:

```
export function setupRouter(..) {
  const router = new Router(
    null, urlSerializer, contexts, location, injector,
    loader, compiler, flatten(config));

  if (urlHandlingStrategy) ..
  if (routeReuseStrategy) ..
  if (opts.errorHandler) ..
  if (opts.enableTracing) ..
  if (opts.onSameUrlNavigation)..
  if (opts.paramsInheritanceStrategy) ..

  return router;
}
```

It instantiates the router service and for each specified option / strategy takes appropriate action. Then it returns the new router service instance. It is important that there is only a single router service per application (the web browser only have a single URL per session) and we need to track how this is so.

`RouterModule` is defined as:

```
@NgModule({declarations: ROUTER_DIRECTIVES, exports: ROUTER_DIRECTIVES})
export class RouterModule {
  // Note: We are injecting the Router so it gets created eagerly...
  constructor(
    @Optional() @Inject(ROUTER_FORROOT_GUARD) guard: any,
    @Optional() router: Router) {}
  static forRoot(routes: Routes, config?: ExtraOptions): ModuleWithProviders
```

```

    { .. }
    static forChild(routes: Routes): ModuleWithProviders { .. }
  }

```

Note that both `forRoot` and `forChild` return a `ModuleWithProviders` instance. What they put in it is different. Recall that this type is defined in:

- [<ANGULAR-MASTER>/packages/core/src/metadata/ng_module.ts](#)

as follows:

```

// wrapper around a module that also includes the providers.
export interface ModuleWithProviders {
  ngModule: Type<any>;
  providers?: Provider[];
}

```

`ForChild` is intended for all except the root routing module. It returns `ngModule` and a list of providers that only contains the result of calling `provideRoutes`:

```

static forChild(routes: Routes): ModuleWithProviders {
  return {ngModule: RouterModule, providers: [provideRoutes(routes)]};
}

```

Critically, it does not contain `ROUTER_PROVIDERS`. In contrast, `forRoot` adds this and many more providers:

```

static forRoot(routes: Routes, config?: ExtraOptions): ModuleWithProviders {
  return {
    ngModule: RouterModule,
    providers: [
      ROUTER_PROVIDERS,
      provideRoutes(routes),
      {provide: ROUTER_FORROOT_GUARD, .. },
      {provide: ROUTER_CONFIGURATION, .. },
      {provide: LocationStrategy, .. },
      {provide: PreloadingStrategy, .. },
      {provide: NgProbeToken, .. },
      provideRouterInitializer(),
    ],
  };
}

```

`ExtraOptions` are additional options passed in to `forRoot` (it is not used with `forChild`):

```

export interface ExtraOptions {
  enableTracing?: boolean;
  useHash?: boolean;
  initialNavigation?: InitialNavigation;
  errorHandler?: ErrorHandler;
  preloadingStrategy?: any;
  onSameUrlNavigation?: 'reload'|'ignore';
  paramsInheritanceStrategy?: 'emptyOnly'|'always';
}

```

For example, if we wished to customize how preloading worked, we need to set the `preloadingStrategy` option.

The `provideRouterInitializer()` function provides a list of initializers:


```

export function provideRouterInitializer() {
  return [
    RouterInitializer,
    {
      provide: APP_INITIALIZER,
      multi: true,
      useFactory: getAppInitializer,
      deps: [RouterInitializer]
    },
    {provide: ROUTER_INITIALIZER,
      useFactory: getBootstrapListener, deps: [RouterInitializer]},
    {provide: APP_BOOTSTRAP_LISTENER, multi: true,
      useExisting: ROUTER_INITIALIZER},
  ];
}

```

This uses the `RouterInitializer` class, whose purpose is best explained by this comment in the code:

```

/**
 * To initialize the router properly we need to do in two steps:
 *
 * We need to start the navigation in a APP_INITIALIZER to block the
 * bootstrap if a resolver or a guards executes asynchronously. Second, we
 * need to actually run activation in a BOOTSTRAP_LISTENER. We utilize the
 * afterPreactivation hook provided by the router to do that.
 *
 * The router navigation starts, reaches the point when preactivation is
 * done, and then pauses. It waits for the hook to be resolved. We then
 * resolve it only in a bootstrap listener.
 */
@Injectable()
export class RouterInitializer { .. }

```

We saw when examining the Core package that its:

- [<ANGULAR_MASTER>/packages/core/src/application_init.ts](#)

defines `APP_INITIALIZER` as:

```

// A function that will be executed when an application is initialized.
export const APP_INITIALIZER =
  new InjectionToken<Array<() => void>>('Application Initializer');

```

`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, currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot
  ): Observable<boolean>|Promise<boolean>|boolean;
}

```

```

}
export interface Resolve<T> {
  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot
): Observable<T>|Promise<T>|T;
}
export interface CanLoad {
  canLoad(route: Route): Observable<boolean>|Promise<boolean>|boolean;
}

```

`router_config_loader.ts` defines a class - `RouterConfigLoader` – and an opaque token, `ROUTES`. After some bookkeeping, `RouterConfigLoader` creates a new instance of `LoadedRouterConfig` it:

```

export const ROUTES = new InjectionToken<Route[] []>('ROUTES');

export class RouterConfigLoader {
  constructor(
    private loader: NgModuleFactoryLoader, private compiler: Compiler,
    private onLoadStartListener?: (r: Route) => void,
    private onLoadEndListener?: (r: Route) => void) {}

  load(parentInjector: Injector, route: Route):
    Observable<LoadedRouterConfig> {
    ..
    return new LoadedRouterConfig(
      flatten(module.injector.get(ROUTES)), module);
  }
}

```

The `router_state.ts` file contains these classes (and some helper functions):

- `RouterState`
- `ActivatedRoute`
- `ActivatedRouteSnapshot`
- `RouterStateSnapshot`

`RouterState` is defined as:

```

// RouterState is a tree of activated routes.
// Every node in this tree knows about the "consumed" URL
// segments, the extracted parameters, and the resolved data.
export class RouterState extends Tree<ActivatedRoute> {
  constructor(
    root: TreeNode<ActivatedRoute>,
    public snapshot: RouterStateSnapshot) {
    super(root);
    setRouterState(<RouterState>this, root);
  }
}

```

`RouterStateSnapshot` is defined as:

```

/**
 * @whatItDoes Represents the state of the router at a moment in time.
 * RouterStateSnapshot is a tree of activated route snapshots. Every node in
 * this tree knows about the "consumed" URL segments, the extracted
 * parameters, and the resolved data.
 */
export class RouterStateSnapshot extends Tree<ActivatedRouteSnapshot> {
  constructor(
    public url: string, root: TreeNode<ActivatedRouteSnapshot>) {

```

```

    super(root);
    setRouterState(<RouterStateSnapshot>this, root);
  }
}

```

The `setRouterStateSnapshot()` function is defined as:

```

function setRouterState<U, T extends { _routerState: U }>(state: U, node:
TreeNode<T>): void {
  node.value._routerState = state;
  node.children.forEach(c => setRouterState(state, c));
}

```

So it sets the router state for the current node, and then recursively calls `setRouterState()` to set it for all children.

The `ActivatedRoute` class is used by the router outlet directive to describe the component it has loaded:

```

// Contains the information about a route associated with a component loaded
// in an outlet. An `ActivatedRoute` can also be used to traverse the
// router state tree
export class ActivatedRoute {
  ..
  constructor(..) {
    this._futureSnapshot = futureSnapshot;
  }

  /** The configuration used to match this route */
  get routeConfig(): Route|null { return this._futureSnapshot.routeConfig; }

  /** The root of the router state */
  get root(): ActivatedRoute { return this._routerState.root; }

  /** The parent of this route in the router state tree */
  get parent(): ActivatedRoute|null {
    return this._routerState.parent(this); }

  /** The first child of this route in the router state tree */
  get firstChild(): ActivatedRoute|null {
    return this._routerState.firstChild(this); }

  /** The children of this route in the router state tree */
  get children(): ActivatedRoute[] {
    return this._routerState.children(this); }

  /** The path from the root of the router state tree to this route */
  get pathFromRoot(): ActivatedRoute[] {
    return this._routerState.pathFromRoot(this); }

  get paramMap(): Observable<ParamMap> { .. }

  get queryParamMap(): Observable<ParamMap> { .. }
}

```

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 {
  @Input() queryParams: {[k: string]: any};
  @Input() fragment: string;
  @Input() queryParamsHandling: QueryParamsHandling;
  @Input() preserveFragment: boolean;
  @Input() skipLocationChange: boolean;
  @Input() replaceUrl: boolean;
  private commands: any[] = [];
  private preserve: boolean;

  constructor(
    private router: Router, private route: ActivatedRoute,
    @Attribute('tabindex') tabIndex: string,
    renderer: Renderer2, el: ElementRef) {

    if (tabIndex == null) {
      renderer.setAttribute(el.nativeElement, 'tabindex', '0');
    }
  }
}
```

The router link commands are set via:

```
@Input()
set routerLink(commands: any[]|string) {
  if (commands != null) {
    this.commands = Array.isArray(commands) ? commands : [commands];
  } else {
    this.commands = [];
  }
}
```

When the link is clicked, the onClick() method is called:

```
@HostListener('click')
onClick(): boolean {
  const extras = {
    skipLocationChange: attrBoolValue(this.skipLocationChange),
    replaceUrl: attrBoolValue(this.replaceUrl),
  };
  this.router.navigateByUrl(this.urlTree, extras);
  return true;
}
```

The urlTree getter uses Router.createUrlTree():

```
get urlTree(): UrlTree {
  return this.router.createUrlTree(this.commands, {
    relativeTo: this.route,
    queryParams: this.queryParams,
    fragment: this.fragment,
    preserveQueryParams: attrBoolValue(this.preserve),
    queryParamsHandling: this.queryParamsHandling,
    preserveFragment: attrBoolValue(this.preserveFragment),
  });
}
```

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.href = this.locationStrategy.prepareExternalUrl(
    this.router.serializeUrl(this.urlTree));
}
```

The router_link_active.ts file contains the RouterLinkActive directive:

```
@Directive({
  selector: '[routerLinkActive]',
  exportAs: '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 defined as:

```
constructor(
  private router: Router,
  private element: ElementRef,
  private renderer: Renderer2,
  private cdr: ChangeDetectorRef) {

  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;
  Promise.resolve().then(() => {
    const hasActiveLinks = this.hasActiveLinks();
    if (this.isActive !== hasActiveLinks) {
      (this as any).isActive = hasActiveLinks;
      this.classes.forEach((c) => {
        if (hasActiveLinks) {
          this.renderer.addClass(this.element.nativeElement, c);
        } else {
          this.renderer.removeClass(this.element.nativeElement, c);
        }
      });
    }
  });
}
```

The `router_outlet.ts` file contains the `RouterOutlet` class:

```
// Acts as a placeholder that Angular dynamically fills based on the
// current router * state.
@Directive({selector: 'router-outlet', exportAs: 'outlet'})
export class RouterOutlet implements OnDestroy, OnInit {
  private activated: ComponentRef<any>|null = null;
  private _activatedRoute: ActivatedRoute|null = null;
  private name: string;

  @Output('activate') activateEvents = new EventEmitter<any>();
  @Output('deactivate') deactivateEvents = new EventEmitter<any>();

  constructor(
    private parentContexts: ChildrenOutletContexts,
    1 private location: ViewContainerRef,
    2 private resolver: ComponentFactoryResolver,
    @Attribute('name') name: string,
    private changeDetector: ChangeDetectorRef) {

    this.name = name || PRIMARY_OUTLET;
    parentContexts.onChildOutletCreated(this.name, this);
  }
}
```

This is where application component whose lifecycle depends on the router live. We note the `ViewContainerRef` **1** and `ComponentFactoryResolver` **2** parameters to the constructor.

`ngOnInit` will either call `attach` **1** or `activateWith` **2**, depending on whether there is an existing component:

```
ngOnInit(): void {
  if (!this.activated) {
    // If the outlet was not instantiated at the time the
    // route got activated we need to populate
    // the outlet when it is initialized (ie inside a NgIf)
    const context = this.parentContexts.getContext(this.name);
    if (context && context.route) {
      if (context.attachRef) {
        // `attachRef` is populated when there is an
        // existing component to mount
        1 this.attach(context.attachRef, context.route);
      } else {
        // otherwise the component defined in the configuration is created
        2 this.activateWith(context.route, context.resolver || null);
      }
    }
  }
}
```

`attach` is defined as:

```
// Called when the `RouteReuseStrategy` instructs to
// re-attach a previously detached subtree
attach(ref: ComponentRef<any>, activatedRoute: ActivatedRoute) {
  this.activated = ref;
  this._activatedRoute = activatedRoute;
  this.location.insert(ref.hostView);
}
```

When its `activateWith` method is called, the resolver will be asked to resolve a component factory for the component:

```
activateWith(
  activatedRoute: ActivatedRoute,
  resolver: ComponentFactoryResolver|null) {
  if (this.isActive) {
    throw new Error('Cannot activate an already activated outlet');
  }
  this._activatedRoute = activatedRoute;
  const snapshot = activatedRoute._futureSnapshot;
  const component = <any>snapshot.routeConfig !.component;
  resolver = resolver || this.resolver;
  const factory = resolver.resolveComponentFactory(component);
  const childContexts =
    this.parentContexts.getOrCreateContext(this.name).children;
  const injector = new OutletInjector(activatedRoute,
    childContexts, this.location.injector);
  this.activated = this.location.createComponent(
    factory, this.location.length, injector);

  // Calling `markForCheck` to make sure we will run the change
  // detection when the // `RouterOutlet` is inside a
  // `ChangeDetectionStrategy.OnPush` component.
  this.changeDetector.markForCheck();
  this.activateEvents.emit(this.activated.instance);
}
}
```

The `location` field is of type `ViewContainerRef`, which we saw being set in the constructor. `ViewContainerRef` is defined in:

- [<ANGULAR-MASTER>/packages/core/src/linker/view_container_ref.ts](#)

```
export abstract class ViewContainerRef {
  // Returns the number of Views currently attached to this container.
  abstract get length(): number;

  // Instantiates a single Component and inserts its Host View into this
  // container at the specified `index`.
  // The component is instantiated using its ComponentFactory which can be
  // obtained via ComponentFactoryResolver.resolveComponentFactory
  // If `index` is not specified, the new View will be inserted as the last
  // View in the container. Returns the {@link ComponentRef} of the Host
  // View created for the newly instantiated Component.
  abstract createComponent<C>(
    componentFactory: ComponentFactory<C>, index?: number,
    injector?: Injector, projectableNodes?: any[][],
    ngModule?: NgModuleRef<any>): ComponentRef<C>;
}
```

So the component is appended as the last entry in the `ViewContainer`.

14: Render3 (Ivy) in Angular

Preliminaries – Paths and names

This document is a guided tour of the new “Ivy” rendering functionality within the modern Angular source tree. We mostly look at the main Angular repository (paths within which we prefix with ANGULAR-MASTER) and also have some discussion of the Angular CLI repository (paths within which we prefix with ANGULAR-CLI-MASTER). You may wish to clone these repositories to your local machine or you may wish to refer to them on Github. If the latter, you should expand these placeholders to:

- ANGULAR-MASTER - <https://github.com/angular/angular>
- ANGUALR-CLI-MASTER - <https://github.com/angular/angular-cli>

We like to use the name “Render3” (instead of “Render 3” with a space), as it helps with Google search, etc. The code-name for this is “Ivy”, we will see that used in places in the code and with the new `enableIvy` option for Compiler-CLI, which can be added via Angular CLI’s `ng new` command, as described here:

- <https://next.angular.io/guide/ivy>

Overview

Rendering in Angular is undergoing further evolution. It seems it was not too long ago that Angular 2’s original Render architecture evolved to Render2, and now along comes the very new Render3, quite a different approach to a view engine.

The main change can be summed up with just this one function (from [<ANGULAR-MASTER>/packages/core/src/render3/interfaces/renderer.ts](https://github.com/angular/angular/blob/master/packages/core/src/render3/interfaces/renderer.ts)):

```
export const domRendererFactory3: RendererFactory3 = {
  createRenderer: (hostElement: RElement | null,
    rendererType: RendererType2 | null):
    Renderer3 => { return document; } ! };
```

By default, the new renderer is compatible with the `document` object from the standard DOM. Actually, it is better not just to say “compatible with” but to say “is”. When running in the browser UI thread, where the DOM is available, then the browser’s native `document` object (provided by the web browser itself, not Angular) really IS the renderer – it is directly used to render content. An app literally cannot perform quicker than that. Regardless of which framework you use and how it structures its rendering pipeline, ultimately this `document` object in the real DOM will have to be called. So why not call it directly in scenarios where it is supported (that means inside the browser UI thread)? That is exactly what Ivy does when possible. For other rendering scenarios (web worker, server, or more specialized, such as WebRTC) then something that looks like the `document` object will need to be provided.

For readers coming from a C++, C#, Java or similar background, it is very important to understand that TypeScript (and JavaScript) uses [structural subtyping](#) (also affectionately called “duck typing”) and not nominal typing (named types) as used by

those other languages. In TypeScript, a type that implements the fields of another type can be used in its place – there is no necessity for implementing common interfaces or to have a common parent class. So the fact that the `document` object from the standard DOM does not implement Angular’s `Render3` but yet is being used in a function to return such a type (see **1** above), is not a problem, so long as it implements all the fields that `Render3` needs (which it does, as we shall soon discover).

Now we will explore in more depth what is happening when we use `enableIvy` with Angular CLI and how the main Angular project delivers `Render3` functionality. We will see three of its packages are involved – `Compiler-CLI`, `Compiler` and `Core`.

Public Documentation

If we visit:

- <https://next.angular.io/api?query=render>

we see the following listing for all parts of the public API that contain the term “render”:

API List

TYPE: All STATUS: All

core

- C** `-RenderComponentType`
- C** `RendererFactory2`
- C** `-RootRenderer`
- C** `-Renderer`
- E** `RendererStyleFlags2`
- C** `Renderer2`
- I** `RendererType2`

core/testing

- C** `TestComponentRenderer`

platform-server

- F** `renderModule`
- F** `renderModuleFactory`

So for now, there is no public API to any `Renderer3` functionality.

The Angular CLI And enableIvy

Application developers wishing to use Ivy will mostly do so via the *enableIvy* command line option to Angular CLI:

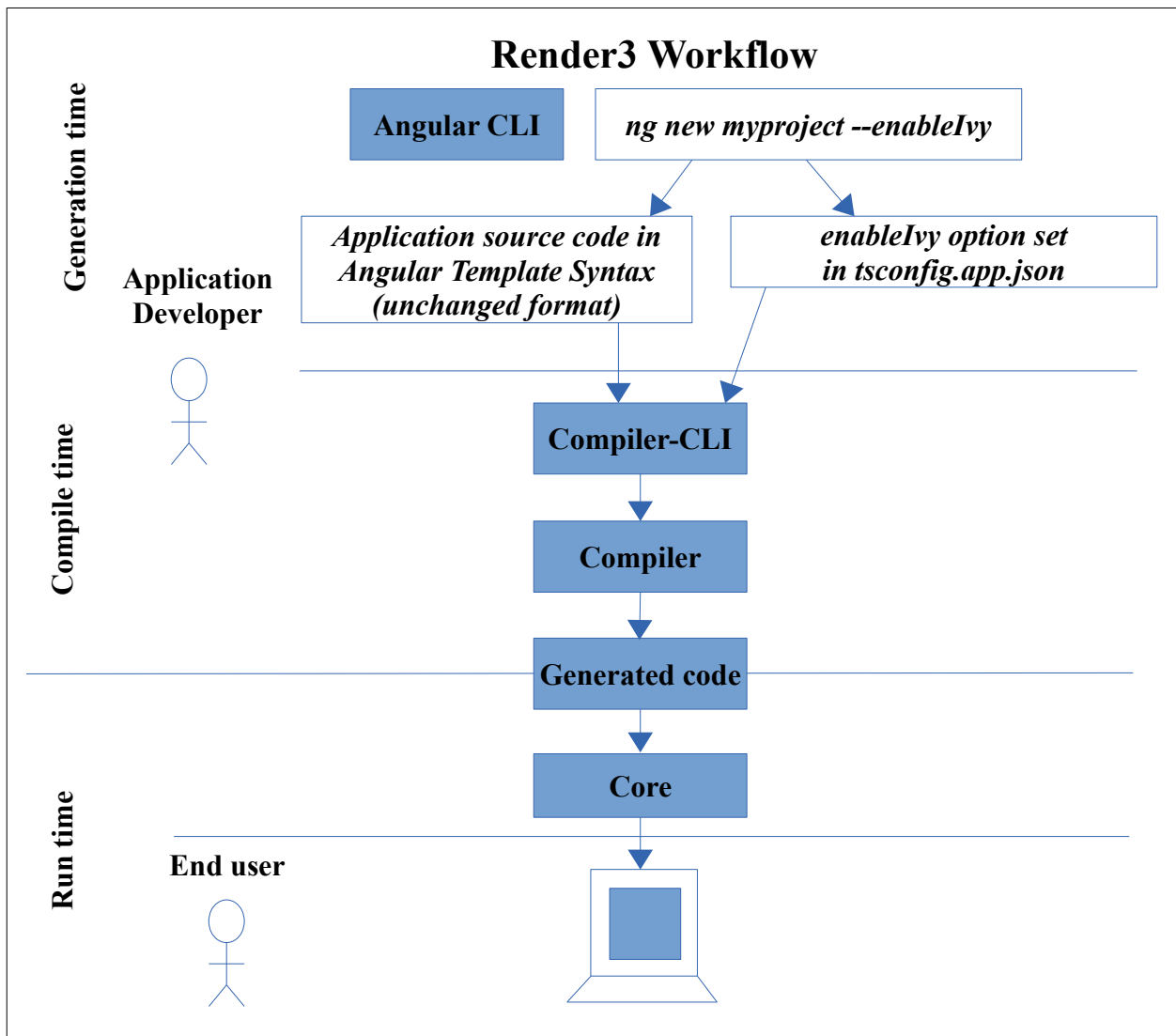
- <https://next.angular.io/guide/ivy>

We can see from the Angular CLI 8.2 source tree this option impacts the code base in a few places. The schema for Angular Application options is defined here:

- [<ANGULAR-CLI-MASTER>/packages/schematics/angular/application/schema.json](https://github.com/angular/angular-cli/blob/master/packages/schematics/angular/application/schema.json)

and includes this for `enableIvy`:

```
{
  "$schema": "http://json-schema.org/schema",
  "id": "SchematicsAngularApp",
  "title": "Angular Application Options Schema",
  "type": "object",
  "description": "Generates a new basic app definition in the \"projects\"
subfolder of the workspace.",
  "properties": {
    "enableIvy": {
      "description": "**EXPERIMENTAL** True to create a new app that
uses the Ivy rendering engine.",
      "type": "boolean",
      "default": false,
      "x-user-analytics": 8
    }
  }
}
```



The entry point for Angular Application schematics:

- [<ANGULAR-CLI-MASTER>/packages/schematics/angular/application/index.ts](#)

has this code :

```
const project = {
  root: normalize(projectRoot),
  sourceRoot,
  projectType: ProjectType.Application,
  prefix: options.prefix || 'app',
  schematics,
  targets: {
    build: {
      builder: Builders.Browser,
      options: {
        ..
        aot: !!options.enableIvy,
```

The schema for Angular Ng New options is defined here:

- [<ANGULAR-CLI-MASTER>/packages/schematics/angular/ng-new/schema.json](#)

and has these entries:

```
{
  "$schema": "http://json-schema.org/schema",
  "id": "SchematicsAngularNgNew",
  "title": "Angular Ng New Options Schema",
  "type": "object",
  "properties": {
    "enableIvy": {
      "description":
        "When true, creates a new app that uses the Ivy rendering engine.",
      "type": "boolean",
      "default": false
    }
  },
}
```

The description string from above ends up here in the generated documentation:

- <https://next.angular.io/cli/new>

The `ng-new` entrypoint:

- [<ANGULAR-CLI-MASTER>/packages/schematics/angular/ng-new/index.ts](#)

accepts the `enableIvy` parameter as follows:

```
const applicationOptions: ApplicationOptions = {
  projectRoot: '',
  name: options.name,
  enableIvy: options.enableIvy,
  ...
}
```

The template for `tsconfig.app.json` reacts to `enableIvy` if present:

- [<ANGULAR-CLI-MASTER>/packages/schematics/angular/application/files/tsconfig.app.json.template](#)

as it has this entry:

```
<% if (enableIvy) { %>
  "angularCompilerOptions": {
    "enableIvy": true
  }<% } %>
```

So that is how the entry gets into `tsconfig.app.json`. Now let's see what impact it has.

The ngtools functionality for webpack configures the bootstrap code slightly differently when `enableIvy` is enabled. In this file:

- [<ANGULAR-CLI-MASTER>/packages/ngtools/webpack/src/transformers/replace_bootstrap.ts](https://github.com/angular/angular-cli/blob/master/packages/ngtools/webpack/src/transformers/replace_bootstrap.ts)

we see:

```
export function replaceBootstrap(
  ..,
  enableIvy?: boolean,
): ts.TransformerFactory<ts.SourceFile> {
  ..
  if (!enableIvy) {
    className += 'NgFactory';
    modulePath += '.ngfactory';
    bootstrapIdentifier = 'bootstrapModuleFactory';
  }
  ..
}
```

So when `enableIvy` is NOT present, the names for the factory artefacts are different. From where does `replaceBootstrap` get called? When we examine:

- [<ANGULAR-CLI-MASTER>/packages/ngtools/webpack/src/angular_compiler_plugin.ts](https://github.com/angular/angular-cli/blob/master/packages/ngtools/webpack/src/angular_compiler_plugin.ts)

we see the `_makeTransformers` method is as follows:

```
private _makeTransformers() {
  ..
  if (this._platformTransformers !== null) {
    this._transformers.push(...this._platformTransformers);
  } else {
    if (this._platform === PLATFORM.Browser) {
      ..
      if (!this._JitMode) {
        // Replace bootstrap in browser AOT.
        this._transformers.push(replaceBootstrap(
          isAppPath,
          getEntryModule,
          getTypeChecker,
          !!this._compilerOptions.enableIvy,
        ));
      }
    } else if (this._platform === PLATFORM.Server) {
      ..
    }
  }
}
```

We also see `ivy` used in `_processLazyRoutes`:

```
// Process the lazy routes discovered, adding them to _lazyRoutes.
// TODO: find a way to remove lazy routes that don't exist anymore.
// This will require a registry of known references to a lazy route,
removing it when no
```

```
// module references it anymore.
private _processLazyRoutes(discoveredLazyRoutes: LazyRouteMap) {
  Object.keys(discoveredLazyRoutes)
    .forEach(lazyRouteKey => {
      ..
      if (this._JitMode ||
        // When using Ivy and not using allowEmptyCodegenFiles,
        // factories are not generated.
        (this._compilerOptions.enableIvy &&
         !this._compilerOptions.allowEmptyCodegenFiles)
      ) {
        modulePath = lazyRouteTSFile;
        moduleKey = `${lazyRouteModule}${moduleName ? '#' + moduleName : ''}`;
      } else { ..

```

We also see ivy impacting on how `_createOrUpdateProgram` works:

```
private async _createOrUpdateProgram() {
  ..
  if (!this.entryModule && !this._compilerOptions.enableIvy) {
    this._warnings.push('Lazy routes discovery is not enabled. '
      + 'Because there is neither an entryModule nor a '
      + 'statically analyzable bootstrap code in the main file.',
    );
  }
}
```

Impact enableIvy has on Angular CLI's code generation

To best see the differences the `enableIvy` option has on code generation, we will now create two new projects – one with and one without the `--enableIvy` option. To save us some time we will use the `--skipInstall` option, which means `npm install` is not run to download all the dependency packages.

```
ng new render3 --enableIvy --skipInstall
ng new render2 --skipInstall
```

A search for `ivy` in the generated `render2` codebase reveals no hits, as expected. A search for `ivy` in the `render3` codebase reveals 2 hits. In `package.json`, `scripts` has this additional item:

```
"scripts": {
  "postinstall": "ivy-ngcc",
  ..
},
```

and in `tsconfig.app.json`, `angularCompilerOptions` has this entry:

```
"angularCompilerOptions": {
  "enableIvy": true
}
```

Now that we have seen how Angular CLI adds `enableIvy`, we are ready to move on to explore how Compiler CLI detects and reacts to this.

Render3 In The Compiler-CLI Package

Despite its name, Compiler CLI is both a set of command-line interfaces and an API (a library). It has three apps, that we see defined in:

- [<ANGULAR-MASTER>/packages/compiler-cli/package.json](#)

as follows, which lists their entry points:

```
"bin": {
  "ngc":      "./src/main.js",
  "ivy-ngcc": "./src/ngcc/main-ngcc.js",
  "ng-xi18n": "./src/extract_i18n.js"
},
```

Ngc is the main Angular template compiler, ivy-ngcc is the Angular Compatibility Compiler and ng-xi18n is for internationalization.

As a library, Compiler CLI mostly exports types from the Compiler package, as seen by its [index.ts](#) file, along with some useful diagnostics types, defined locally within Compiler CLI. The naming of none of these exports is Ivy-specific, but their internals are impacted by use of Ivy, as we shall soon see.

In root of the src tree, there are references to Ivy in main.ts and perform_compile.ts.

[main.ts](#) has this:

```
function createEmitCallback(options: api.CompilerOptions):
    api.TsEmitCallback|undefined {
  const transformDecorators =
    !options.enableIvy && options.annotationsAs !== 'decorators';
```

and this detailed error reporter:

```
function reportErrorsAndExit(
  allDiagnostics: Diagnostics, options?: api.CompilerOptions,
  consoleError: (s: string) => void = console.error): number {
  const errorsAndWarnings = filterErrorsAndWarnings(allDiagnostics);
  if (errorsAndWarnings.length) {
    const formatHost = getFormatDiagnosticsHost(options);
    if (options && options.enableIvy === true) {
      const ngDiagnostics = errorsAndWarnings.filter(api.isNgDiagnostic);
      const tsDiagnostics = errorsAndWarnings.filter(api.isTsDiagnostic);
      consoleError(replaceTsWithNgInErrors(
        ts.formatDiagnosticsWithColorAndContext(
          tsDiagnostics, formatHost)));
      consoleError(formatDiagnostics(ngDiagnostics, formatHost));
    } else {
      consoleError(formatDiagnostics(errorsAndWarnings, formatHost));
    }
  }
  return exitCodeFromResult(allDiagnostics);
}
```

[perform_compile.ts](#) has this:

```
export function createNgCompilerOptions(
  basePath: string, config: any, tsOptions: ts.CompilerOptions):
  api.CompilerOptions {
  // enableIvy `ngtsc` is an alias for `true`.
  if (config.angularCompilerOptions &&
      config.angularCompilerOptions.enableIvy === 'ngtsc') {
    config.angularCompilerOptions.enableIvy = true;
  }
  return {...tsOptions, ...config.angularCompilerOptions,
          genDir: basePath, basePath};
}
```

The CompilerOptions interface is defined in the transformers sub-directory. In:

- [/packages/compiler-cli/src/transformers/api.ts](https://github.com/angular/angular/blob/master/packages/compiler-cli/src/transformers/api.ts)

we see it defines an interface, CompilerOptions, that extends ts.CompilerOptions with Angular-specific fields.

```
export interface CompilerOptions extends ts.CompilerOptions {
  // NOTE: These comments and aio/content/guides/aot-compiler.md
  // should be kept in sync.
  ..
}
```

Despite this comment about aot-compiler.md, that Markdown file actually contains no mention of Ivy. The CompilerOptions interface in api.ts does contain this addition:

```
/**
 * Tells the compiler to generate definitions using the Render3 style code
 * generation. This option defaults to `false`.
 *
 * Not all features are supported with this option enabled. It is only
 * supported for experimentation and testing of Render3 style code
 * generation.
 * Acceptable values are as follows:
 *
 * `false` - run ngc normally
 * `true` - run the ngtsc compiler instead of the normal ngc compiler
 * `ngtsc` - alias for `true`
 * `tsc` - behave like plain tsc as much as possible
 *          (used for testing JIT code)
 *
 * @publicApi
 */
enableIvy?: boolean|'ngtsc'|'tsc';
}
```

So we see there is a new ngtsc compiler in addition to the normal tsc compiler, and this switch is used to select one or the other. We will soon explore how ngtsc works.

The api.ts file also defines EmitFlags, which is of interest to us (note the default includes codegen):

```
export enum EmitFlags {
  DTS = 1 << 0,
  JS = 1 << 1,
  Metadata = 1 << 2,
  I18nBundle = 1 << 3,
  Codegen = 1 << 4,
```



```

    Default = DTS | JS | Codegen,
    All = DTS | JS | Metadata | I18nBundle | Codegen,
  }

```

The `tsc_pass_through.ts` file defines an implementation of the Program API:

```

import {ivySwitchTransform} from '../ngtsc/switch';

/**
 * An implementation of the `Program` API which behaves similarly to plain `
 * tsc`. The only Angular specific behavior included in this `Program` is the
 * operation of the Ivy switch to turn on render3 behavior. This allows `ngc`
 * to behave like `tsc` in cases where JIT code needs to be tested.
 */
export class TscPassThroughProgram implements api.Program {
  ...
  emit(opts?: { .. }): ts.EmitResult {
    const emitCallback = opts && opts.emitCallback || defaultEmitCallback;
    const emitResult = emitCallback({
      ..
      customTransformers: {before: [ivySwitchTransform]},
    });
    return emitResult;
  }
}

```

We see Ivy impacting the `program.ts` file in a number of ways.

- [<ANGULAR-MASTER>/packages/compiler-cli/src/transformers/program.ts](https://github.com/angular/angular/blob/master/packages/compiler-cli/src/transformers/program.ts)

The command line option is extracted in `getAotCompilerOptions()`:

```

// Compute the AotCompiler options
function getAotCompilerOptions(options: CompilerOptions): AotCompilerOptions
{
  ..
  return {
    ..
    enableIvy: options.enableIvy
  };
}

```

Metadata is expected to be lower case:

```

// Fields to lower within metadata in render2 mode.
const LOWER_FIELDS =
  ['useValue', 'useFactory', 'data', 'id', 'loadChildren'];
// Fields to lower within metadata in render3 mode.
const R3_LOWER_FIELDS = [...LOWER_FIELDS, 'providers', 'imports', 'exports'];

class AngularCompilerProgram implements Program {
  constructor(
    ...
    this.loweringMetadataTransform =
      new LowerMetadataTransform(
        options.enableIvy ? R3_LOWER_FIELDS : LOWER_FIELDS);

```

Also in `AngularCompilerProgram` we see the use of reified decorators for Render3:

```

const R3_REIFIED_DECORATORS = [
  'Component',

```

```

    'Directive',
    'Injectable',
    'NgModule',
    'Pipe',
  ];

  private get reifiedDecorators(): Set<StaticSymbol> {
    if (!this._reifiedDecorators) {
      const reflector = this.compiler.reflector;
      this._reifiedDecorators = new Set(
        R3_REIFIED_DECORATORS.map(
          name => reflector.findDeclaration('@angular/core', name));
    }
    return this._reifiedDecorators;
  }
}

```

The emit method has protection against being inadvertently called for Ivy:

```

emit(parameters: {
  emitFlags?: EmitFlags,
  cancellationToken?: ts.CancellationToken,
  customTransformers?: CustomTransformers,
  emitCallback?: TsEmitCallback,
  mergeEmitResultsCallback?: TsMergeEmitResultsCallback,
} = {}): ts.EmitResult {
  if (this.options.enableIvy) {
    throw new Error('Cannot run legacy compiler in ngtsvc mode!');
  }
  return this._emitRender2(parameters);
}

```

The createProgram function is where the decision is made which compilation program to use; this is influenced by the value for enableIvy:

```

export function createProgram({rootNames, options, host, oldProgram}: {
  rootNames: ReadonlyArray<string>,
  options: CompilerOptions,
  host: CompilerHost, oldProgram?: Program
}): Program {
  if (options.enableIvy === true) {
    return new NgtsvcProgram(rootNames, options, host, oldProgram);
  } else if (options.enableIvy === 'tsc') {
    return new TscPassThroughProgram(rootNames, options, host, oldProgram);
  }
  return new AngularCompilerProgram(rootNames, options, host, oldProgram);
}

```

Before leaving program.ts, we wish to mention two other functions. This file also contains the defaultEmitCallback:

```

const defaultEmitCallback: TsEmitCallback =
  ({program, targetSourceFile, writeFile, cancellationToken,
    emitOnlyDtsFiles, customTransformers}) =>
  1   program.emit(
    targetSourceFile, writeFile, cancellationToken,
    emitOnlyDtsFiles, customTransformers);

```

We see at **1** where the compilation is actually initiated with the set of custom transformers which was passed in as a parameter.

as:

```
export class PartialModuleMetadataTransformer implements MetadataTransformer
{
  private moduleMap: Map<string, PartialModule>;

  constructor(modules: PartialModule[]) {
    this.moduleMap = new Map(modules.map<[string, PartialModule]>(
      m => [m.fileName, m]));
  }

  start(sourceFile: ts.SourceFile): ValueTransform|undefined {
    const partialModule = this.moduleMap.get(sourceFile.fileName);
    if (partialModule) {
      const classMap = new Map<string, ClassStmt>(
        partialModule.statements.filter(isClassStmt)
          .map<[string, ClassStmt]>(s => [s.name, s]));
      if (classMap.size > 0) {
        return (value: MetadataValue, node: ts.Node): MetadataValue => {
          // For class metadata that is going to be transformed to have a
          // static method ensure the metadata contains a
          // static declaration the new static method.
          if (isClassMetadata(value) &&
            node.kind === ts.SyntaxKind.ClassDeclaration) {
            const classDeclaration = node as ts.ClassDeclaration;
            if (classDeclaration.name) {
              const partialClass = classMap.get(classDeclaration.name.text);
              if (partialClass) {
                for (const field of partialClass.fields) {
                  if (field.name && field.modifiers &&
                    field.modifiers.some(modifier =>
                      modifier === StmtModifier.Static)) {
                    value.statics =
                      {...(value.statics || {}), [field.name]: {}};
                  }
                }
              }
            }
          }
          return value;
        };
      }
    }
  }
}
```

Now we are ready go back to Compiler-CLI's [program.ts](#) file to look at:

```
private _emitRender3(
  1 {emitFlags = EmitFlags.Default, cancellationToken, customTransformers,
  2   emitCallback = defaultEmitCallback}: {
    emitFlags?: EmitFlags,
    cancellationToken?: ts.CancellationToken,
    customTransformers?: CustomTransformers,
    emitCallback?: TsEmitCallback
  3 } = {}): ts.EmitResult {
  const emitStart = Date.now();

  // .. Check emitFlags
  // .. Set up code to emit partical modules
```

```

    // .. Set up code for file writing (not shown)
    // .. Build list of custom transformers
    // .. Make emitResult
    return emitResult;
}

```

This takes a range of input parameters (note `emitFlags` **1** and `emitCallback` **2**) and then returns an instance of `ts.EmitResult` **3**. The `emitFlags` says what needs to be emitted – if nothing, then we return immediately:

```

    if ((emitFlags &
        (EmitFlags.JS | EmitFlags.DTS | EmitFlags.Metadata | EmitFlags.Codegen))
        === 0) {
        return {emitSkipped: true, diagnostics: [], emittedFiles: []};
    }

```

The partial modules are processed with this important call to `emitAllPartialModules` in Angular’s Compiler package (we will examine this in detail shortly):

```
const modules = this.compiler.emitAllPartialModules(this.analyzedModules);
```

The `this.analyzedModules` getter is defined as:

```

private get analyzedModules(): NgAnalyzedModules {
    if (!this._analyzedModules) {
        this.initSync();
    }
    return this._analyzedModules !; 1
}

```

Note the `!` at the end of the return **1**: this is the TypeScript non-null assertion operator, a new language feature explained as:

“A new `!` post-fix expression operator may be used to assert that its operand is non-null and non-undefined in contexts where the type checker is unable to conclude that fact. Specifically, the operation `x!` produces a value of the type of `x` with null and undefined excluded.” ([TypeScript release notes](#))

The `_analyzedModules` field is initialized earlier as:

```
private _analyzedModules: NgAnalyzedModules|undefined;
```

Returning to our coverage of `_emitRender3` – after the call to `this.compiler.emitAllPartialModules` to emit the modules, the `writeTsFile` and `emitOnlyDtsFiles` consts are set up:

```

const writeTsFile: ts.WriteFileCallback =
    (outFileName, outData, writeByteOrderMark, onError?, sourceFiles?)
    => {
        const sourceFile = sourceFiles && sourceFiles.length
                                == 1 ? sourceFiles[0] : null;
        let genFile: GeneratedFile|undefined;
        this.writeFile(outFileName, outData, writeByteOrderMark,
                                onError, undefined, sourceFiles);
    };

const emitOnlyDtsFiles =
    (emitFlags & (EmitFlags.DTS | EmitFlags.JS)) == EmitFlags.DTS;

```

Then the custom transformers are configured (note the partial modules parameter):

```
const tsCustomTransformers = this.calculateTransforms(
  /* genFiles */ undefined,
  /* partialModules */ modules, customTransformers);
```

Finally the `emitResult` is set up like so (note the `customTransformers` in there) and then returned:

```
const emitResult = emitCallback({
  program: this.tsProgram,
  host: this.host,
  options: this.options,
  writeFile: writeTsFile, emitOnlyDtsFiles,
  customTransformers: tsCustomTransformers
});

return emitResult;
```

We should briefly mention differences between `_emitRender3` and `_emitRender2`, also in [program.ts](#). The latter is a much larger function compared to `_emitRender3`:

```
private _emitRender2(
  {emitFlags = EmitFlags.Default, cancellationToken, customTransformers,
  emitCallback = defaultEmitCallback}: {
    emitFlags?: EmitFlags,
    cancellationToken?: ts.CancellationToken,
    customTransformers?: CustomTransformers,
    emitCallback?: TsEmitCallback
  } = {}): ts.EmitResult {
  ..
  let {genFiles, genDiags} = this.generateFilesForEmit(emitFlags);
  ..
}
```

It also used a different helper function, `generateFilesForEmit`, to make a different call into Angular's Compiler package. So importantly we have two separate call paths into the Angular Compiler package depending on which renderer we are using:

```
private generateFilesForEmit(emitFlags: EmitFlags):
  {genFiles: GeneratedFile[], genDiags: ts.Diagnostic[]} {
  ..
  let genFiles = this.compiler.emitAllImpls(this.analyzedModules).filter(
    genFile => isInRootDir(genFile.genFileUrl, this.options));

  return {genFiles, genDiags: []};
}
```

The ivy-ngcc tool called here is the Angular Compatibility Compiler from Compiler-CLI which is described here:

- [<ANGULAR-MASTER>/packages/compiler-cli/src/ngcc](https://github.com/angular/angular/blob/master/packages/compiler-cli/src/ngcc)

as follows:

This compiler will convert node_modules compiled with ngc, into node_modules which appear to have been compiled with ngts. This conversion will allow such "legacy" packages to be used by the Ivy rendering engine.

There is a mention of ngcc here:

- <https://next.angular.io/guide/ivy#ngcc>

When exploring Compiler-CLI soon, we will look at ngcc.

Render3 in The Compiler Package

Inside the Angular Compiler package:

- [<ANGULAR_MASTER>/packages/compiler](#)

the Render3 feature resides mainly in four files. They are:

- [<ANGULAR-MASTER>/packages/compiler/src/aot/partial_module.ts](#)
- [<ANGULAR-MASTER>/packages/compiler/src/aot/compiler.ts](#)
- [<ANGULAR-MASTER>/packages/compiler/src/render3/r3_identifiers.ts](#)
- [<ANGULAR-MASTER>/packages/compiler/src/render3/r3_view_compiler.ts](#)

Let's start with `partial_module.ts`. It has these few lines, to describe what a partial module type is:

```
import * as o from '../output/output_ast';

export interface PartialModule {
  fileName: string;
  statements: o.Statement[];
}
```

We have seen from our coverage of Compiler-CLI that it makes a call to `emitAllPartialModules` inside the Compiler package. This is to be found in the [src/aot/compiler.ts](#) file and so is exported by this line from [src/compiler.ts](#) (yes: same files names, different directories):

```
export * from './aot/compiler';
```

it is defined as:

```
emitAllPartialModules({ngModuleByPipeOrDirective, files}:
NgAnalyzedModules): PartialModule[] {
  // Using reduce like this is a select
  // many pattern (where map is a select pattern)
  return files.reduce<PartialModule[]>((r, file) => {
    r.push(...this._emitPartialModule(
      file.fileName, ngModuleByPipeOrDirective, file.directives,
      file.pipes, file.ngModules, file.injectables));
    return r;
  }, []);
}
```

It calls the internal `_emitPartialModule` method:

```
private _emitPartialModule(
  fileName: string,
  ngModuleByPipeOrDirective: Map<StaticSymbol, CompileNgModuleMetadata>,
  directives: StaticSymbol[],
  pipes: StaticSymbol[],
  ngModules: CompileNgModuleMetadata[],
  injectables: StaticSymbol[]): PartialModule[] {

  const classes: o.ClassStmt[] = [];
  const context = this._createOutputContext(fileName);
```

After initializing context information, it loops **1** over the directives array and if a component is found **2**, calls `compileIvyComponent` **2**, otherwise calls `compileIvyDirective` **3**:


```

// Process all components and directives
1 directives.forEach(directiveType => {
    const directiveMetadata =
      this._metadataResolver.getDirectiveMetadata(directiveType);
    if (directiveMetadata.isComponent) {
      ..
      const {template: parsedTemplate} =
        this._parseTemplate(
          directiveMetadata, module, module.transitiveModule.directives);
2     compileIvyComponent(
        context, directiveMetadata, parsedTemplate, this._reflector);
    } else {
3     compileIvyDirective(context, directiveMetadata, this._reflector);
    }
  });

  if (context.statements) {
    return [{fileName, statements:
      [...context.constantPool.statements, ...context.statements]};
  }
  return [];
}

```

We note the import at the top of the file:

```

import {
  compileComponent as compileIvyComponent,
  compileDirective as compileIvyDirective}
  from '../render3/r3_view_compiler';

```

So in [src/render3/r3_view_compiler.ts](#) let's track `compileComponent` and `compileDirective`.

The [render3 sub-directory](#) in the Compiler package's `src` directory is new for Render3. It contains just two files, `r3_view_compiler` and `r3_identifiers.ts`. `r3_identifiers.ts` is imported into `r3_view_compiler.ts` with this line:

```

import {Identifiers as R3} from './r3_identifiers';

```

So anywhere in `r3_view_compiler.ts` we see "R3" being used for naming (over 50 times), it means something in `r3_identifiers.ts` is being used. `r3_identifiers.ts` is not referenced from anywhere else in the Compiler package.

`r3_identifier.ts` contains a long list of external reference identifiers for the various instructions. Here is a sampling (note that "o" is imported from `output_ast.ts`):

```

import * as o from '../output/output_ast';
const CORE = '@angular/core';
export class Identifiers {
  /* Methods */
  static NEW_METHOD = 'n';
  static HOST_BINDING_METHOD = 'h';

  /* Instructions */
  static createElement: o.ExternalReference = {name: 'eE', moduleName: CORE};
  static elementEnd: o.ExternalReference = {name: 'ee', moduleName: CORE};
  static text: o.ExternalReference = {name: 'eT', moduleName: CORE};
  static bind: o.ExternalReference = {name: 'eb', moduleName: CORE};
  static bind1: o.ExternalReference = {name: 'eb1', moduleName: CORE};

```

```

static bind2: o.ExternalReference = {name: 'eb2', moduleName: CORE};
static projection: o.ExternalReference = {name: 'eP', moduleName: CORE};
static projectionDef: o.ExternalReference =
    {name: 'epD', moduleName: CORE};
static injectElementRef: o.ExternalReference =
    {name: 'einjectElementRef', moduleName: CORE};
static injectTemplateRef: o.ExternalReference =
    {name: 'einjectTemplateRef', moduleName: CORE};
static defineComponent: o.ExternalReference =
    {name: 'edefineComponent', moduleName: CORE};
    ..
};

```

The `compileDirective` function is implemented in [src/render3/r3_view_compiler.ts](#) as:

```

export function compileDirective(
    outputCtx: OutputContext,
    directive: CompileDirectiveMetadata,
    reflector: CompileReflector) {

    const definitionMapValues:
        {key: string, quoted: boolean, value: o.Expression}[] = [];

    // e.g. 'type: MyDirective`
    definitionMapValues.push(
        {key: 'type',
         value: outputCtx.importExpr(directive.type.reference),
         quoted: false});

    // e.g. `factory: () => new MyApp(injectElementRef())`
    1 const templateFactory =
        createFactory(directive.type, outputCtx, reflector);
    2 definitionMapValues.push(
        {key: 'factory', value: templateFactory, quoted: false});

    // e.g 'inputs: {a: 'a'}`
    if (Object.getOwnPropertyNames(directive.inputs).length > 0) {
        definitionMapValues.push(
            {key: 'inputs',
             quoted: false,
             value: mapToExpression(directive.inputs)});
    }

    const className = identifierName(directive.type) !;
    className || error(`Cannot resolver the name of ${directive.type}`);

    // Create the partial class to be merged with the actual class.
    3 outputCtx.statements.push( 4 new o.ClassStmt(
        /* name */ className,
        /* parent */ null,
        /* fields */[new o.ClassField(
            /* name */ 'ngDirectiveDef',
            /* type */ o.INFERRED_TYPE,
            /* modifiers */[o.StmtModifier.Static],
            /* initializer */ 5 o.importExpr(R3.defineDirective).callFn(
                [o.literalMap(definitionMapValues)])],
        /* getters */[],
        /* constructorMethod */ new o.ClassMethod(null, [], []),
        /* methods */[]));

```

```
}

```

It first **1** creates a template factory and then pushes it on the definition map values array **2**. Then it uses the output context **3** to push a new `Class` statement **4** onto the array of statements. We note the initializer is set to `R3.defineDirective` **5**.

The `compileComponent` function (also in [r3_view_compiler.ts](#)) is a little bit more complex. Let's look at it in stages. Its signature is:

```
export function compileComponent(
  outputCtx: OutputContext,
  component: CompileDirectiveMetadata,
  template: TemplateAst[],
  reflector: CompileReflector) {

  const definitionMapValues:
    {key: string, quoted: boolean, value: o.Expression}[] = [];
  // e.g. `type: MyApp`
  definitionMapValues.push(
    {key: 'type',
     value: outputCtx.importExpr(component.type.reference),
     quoted: false});
  ... // some code regarding selectors (omitted)

```

Then it sets up a template function expression on the definition map values:

```
// e.g. `factory: function MyApp_Factory()
// { return new MyApp(injectElementRef()); }`
const templateFactory = 1 createFactory(
  component.type, outputCtx, reflector);
definitionMapValues.push({
  key: 'factory',
  value: templateFactory,
  quoted: false});

```

We note the call to the `createFactory` function **1**, which we need to follow up in a bit. Then it sets up a template definition builder and again adds it to the definition map values array:

```
// e.g. `template: function MyComponent_Template(_ctx, _cm) {...}`
const templateTypeName = component.type.reference.name;
const templateName =
  templateTypeName ? `${templateTypeName}_Template` : null;

const templateFunctionExpression =
1 new TemplateDefinitionBuilder(
  outputCtx, outputCtx.constantPool, reflector, CONTEXT_NAME,
  ROOT_SCOPE.nestedScope(), 0, component.template!.ngContentSelectors,
  templateTypeName, templateName)
2 .buildTemplateFunction(template, []);
definitionMapValues.push({
  key: 'template',
  value: templateFunctionExpression,
  quoted: false});

```

We note the use of the `TemplateDefinitionBuilder` class **1**, and the call to its `buildTemplateFunction` method **2**, both of which we will examine shortly. Then it sets up the class name (and uses the `!` non-null assertion operator to ensure it is not null):

```
const className = identifierName(component.type) !;
```

Finally it adds the new class statement:

```
// Create the partial class to be merged with the actual class.
outputCtx.statements.push(new o.ClassStmt (
  /* name */ className,
  /* parent */ null,
  /* fields */[new o.ClassField(
    /* name */ 'ngComponentDef',
    /* type */ o.INFERRED_TYPE,
    /* modifiers */[o.StmtModifier.Static],
    /* initializer */ 1 o.importExpr(R3.defineComponent).callFn(
      [o.literalMap(definitionMapValues)])),
  /* getters */[],
  /* constructorMethod */ new o.ClassMethod(null, [], []),
  /* methods */[]));
```

We note the initializer is set to `R3.defineComponent` **1**.

The `createFactory` function is defined as:

```
function createFactory(
  type: CompileTypeMetadata, outputCtx: OutputContext,
  reflector: CompileReflector): o.FunctionExpr {
  let args: o.Expression[] = [];
```

It first resolves three reflectors:

```
const elementRef =
  reflector.resolveExternalReference(Identifiers.ElementRef);
const templateRef =
  reflector.resolveExternalReference(Identifiers.TemplateRef);
const viewContainerRef =
  reflector.resolveExternalReference(Identifiers.ViewContainerRef);
```

Then it loops through the `type.diDeps` dependencies, and pushes a relevant import expression, based on the token ref:

```
for (let dependency of type.diDeps) {
  if (dependency.isValue) {
    unsupported('value dependencies');
  }
  if (dependency.isHost) {
    unsupported('host dependencies');
  }
  const token = dependency.token;
  if (token) {
    const tokenRef = tokenReference(token);
    if (tokenRef === elementRef) {
      args.push(o.importExpr(R3.injectElementRef).callFn([]));
    } else if (tokenRef === templateRef) {
      args.push(o.importExpr(R3.injectTemplateRef).callFn([]));
    } else if (tokenRef === viewContainerRef) {
      args.push(o.importExpr(R3.injectViewContainerRef).callFn([]));
    } else {
      const value =
        token.identifier != null ?
          outputCtx.importExpr(tokenRef) : o.literal(tokenRef);
      args.push(o.importExpr(R3.inject).callFn([value]));
    }
  }
}
```

```

    } else {
      unsupported('dependency without a token');
    }
  }

  return o.fn(
    [],
    [new o.ReturnStatement(
      new o.InstantiateExpr(outputCtx.importExpr(type.reference), args)],
    o.INFERRED_TYPE, null, type.reference.name ?
      `${type.reference.name}_Factory` : null);
  )
}

```

The `TemplateDefinitionBuilder` class (also located in [r3_view_compiler.ts](#)) is large (350 lines+) and can be considered the heart of Render3 compilation. It implements the `TemplateAstVisitor` interface. This interface is defined in:

- [<ANGULAR-MASTER>/packages/compiler/src/template_parser/template_ast.ts](#)

as follows:

```

// A visitor for {@link TemplateAst} trees that will process each node.
export interface TemplateAstVisitor {
  visit?(ast: TemplateAst, context: any): any;
  visitNgContent(ast: NgContentAst, context: any): any;
  visitEmbeddedTemplate(ast: EmbeddedTemplateAst, context: any): any;
  visitElement(ast: ElementAst, context: any): any;
  visitReference(ast: ReferenceAst, context: any): any;
  visitVariable(ast: VariableAst, context: any): any;
  visitEvent(ast: BoundEventAst, context: any): any;
  visitElementProperty(ast: BoundElementPropertyAst, context: any): any;
  visitAttr(ast: AttrAst, context: any): any;
  visitBoundText(ast: BoundTextAst, context: any): any;
  visitText(ast: TextAst, context: any): any;
  visitDirective(ast: DirectiveAst, context: any): any;
  visitDirectiveProperty(ast: BoundDirectivePropertyAst, context: any): any;
}

```

Back in [r3_view_compiler.ts](#), the definition of `TemplateDefinitionBuilder` begins with:

```

class TemplateDefinitionBuilder
  implements TemplateAstVisitor, LocalResolver {

  constructor(
    private outputCtx: OutputContext,
    private constantPool: ConstantPool,
    private reflector: CompileReflector,
    private contextParameter: string,
    private bindingScope: BindingScope,
    private level = 0,
    private ngContentSelectors: string[],
    private contextName: string|null,
    private templateName: string|null) {}
}

```

We saw the call to `buildTemplateFunction` early in `compileComponent` – its has this signature:

```

buildTemplateFunction(
  asts: TemplateAst[], variables: VariableAst[]): o.FunctionExpr {

```

It returns an instance of `o.FunctionExpr`. We note the import at the top of the file:

```
import * as o from '../output/output_ast';
```

So `o.FunctionExpr` means the `FunctionExpr` class in:

- [<ANGULAR-MASTER>/packages/compiler/src/output/output_ast.ts](#)

This class is defined as follows:

```
export class FunctionExpr extends Expression {
  constructor(
    public params: FnParam[],
    public statements: Statement[],
    type?: Type|null,
    sourceSpan?: ParseSourceSpan|null,
    public name?: string|null) {

    super(type, sourceSpan);
  }
  ...
}
```

While we are looking at `output_ast.ts`, we see this `fn` function:

```
export function fn(
  params: FnParam[],
  body: Statement[], type?: Type | null,
  sourceSpan?: ParseSourceSpan | null,
  name?: string | null): FunctionExpr {

  return new FunctionExpr(params, body, type, sourceSpan, name);
}
```

It just makes a `FunctionExpr` from the supplied parameters.

An interesting function in [src/template_parser/template_ast.ts](#) is `templateVisitAll`:

```
/**
 * Visit every node in a list of {@link TemplateAst}s with the given
 * {@link TemplateAstVisitor}.
 */
export function templateVisitAll(
  visitor: TemplateAstVisitor,
  asts: TemplateAst[],
  context: any = null): any[] {

  const result: any[] = [];
  const visit = visitor.visit ?
    (ast: TemplateAst) => visitor.visit !(ast, context) ||
  ast.visit(visitor, context) :
    (ast: TemplateAst) => ast.visit(visitor, context);
  asts.forEach(ast => {
    const astResult = visit(ast);
    if (astResult) {
      result.push(astResult);
    }
  });
  return result;
}
```

Now let's return to the critically important `buildTemplateFunction` method of `TemplateDefinitionBuilder` in [r3_view_compiler.ts](#) – a summary of its definition is:

```

buildTemplateFunction(
  asts: TemplateAst[],
  variables: VariableAst[]): o.FunctionExpr {

  // Create variable bindings
  ...
  // Collect content projections
  ...
1  templateVisitAll(this, asts);
  ..
2  return o.fn(
    [
3    new o.FnParam(this.contextParameter, null),
      new o.FnParam(CREATION_MODE_FLAG, o.BOOL_TYPE)
    ],
    [
4    // Temporary variable declarations (i.e. let _t: any;)
      ...this._prefix,

      // Creating mode (i.e. if (cm) { ... })
      ...creationMode,

      // Binding mode (i.e. ep(...))
      ...this._bindingMode,

      // Host mode (i.e. Comp.h(...))
      ...this._hostMode,

      // Refresh mode (i.e. Comp.r(...))
      ...this._refreshMode,

      // Nested templates (i.e. function CompTemplate() {})
      ...this._postfix
    ],
5    o.INFERRED_TYPE, null, this.templateName);
  }

```

We see it first visits the template tree **1**. Then it returns the result of a call **2** to the `fn` function we just looked at, passing in three entries – **3** an array of `FnParams`, **4** an array of statements and **5** `o.INFERRED_TYPE`. What is happening here is that each node in the template tree is being visited, and where appropriate, statements are being emitted to the output statement array with the correct Render3 instruction. The instruction function is used to add a statement like so:

```

private instruction(
  statements: o.Statement[],
  span: ParseSourceSpan|null,
  reference: o.ExternalReference,
  ...params: o.Expression[]) {
  statements.push(
    o.importExpr(reference, null, span).callFn(params, span).toStmt());
}

```

For example, when a text node is visited, the Render3 text instruction (`R3.text`) should be emitted. We see this happening with the `visitText` method:

```
private _creationMode: o.Statement[] = [];

visitText(ast: TextAst) {
  // Text is defined in creation mode only.
  this.instruction(
    this._creationMode, ast.sourceSpan, R3.text,
    o.literal(this.allocateDataSlot()), o.literal(ast.value));
}
```

There is an equivalent method for elements, `visitElement`, which is somewhat more complex. After some setup code, it has this:

```
// Generate the instruction create element instruction
this.instruction(this._creationMode, ast.sourceSpan,
  R3.createElement, ...parameters);
```

There is also a `visitEmbeddedTemplate` method, which emits a number of Render3 instructions:

```
visitEmbeddedTemplate(ast: EmbeddedTemplateAst) {
  ...
  // e.g. C(1, C1Template)
  this.instruction(
    this._creationMode, ast.sourceSpan,
    R3.containerCreate, o.literal(templateIndex),
    directivesArray, o.variable(templateName));

  // e.g. Cr(1)
  this.instruction(
    this._refreshMode, ast.sourceSpan,
    R3.containerRefreshStart, o.literal(templateIndex));

  // Generate directives
  this._visitDirectives(
    ast.directives, o.variable(this.contextParameter),
    templateIndex, directiveIndexMap);

  // e.g. cr();
  this.instruction(
    this._refreshMode, ast.sourceSpan, R3.containerRefreshEnd);

  // Create the template function
  const templateVisitor = new TemplateDefinitionBuilder(
    this.outputCtx, this.constantPool, this.reflector, templateContext,
    this.bindingScope.nestedScope(), this.level + 1,
    this.ngContentSelectors, contextName, templateName);
  const templateFunctionExpr = templateVisitor.buildTemplateFunction(
    ast.children, ast.variables);
  this._postfix.push(templateFunctionExpr.toDeclStmt(templateName, null));
}
```


Render3 in the Core Package

API Model

There is no public API to Render3. The [Core package](#) contains the Render3 (and Render2) code. Its [index.ts](#) file just exports the contents of [public_api.ts](#), which in turn exports the contents of [./src/core.ts](#).

Regarding the public API, this has one render-related line, an export of:

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

The [./src/render.ts](#) file exports no Render3 API. It does export the Render2 API, like so:

```
export {RenderComponentType, Renderer, Renderer2, RendererFactory2,
  RendererStyleFlags2, RendererType2, RootRenderer} from './render/api';
```

Note that Render2 is just the [./src/render/api.ts](#) file with less than 200 lines of code (the [core/src/render](#) sub-directory only contains that one file)- it defines the above types but does not contain an implementation. You can read it in full here:

- [<ANGULAR-MASTER>/packages/core/src/render/api.ts](#)

Render3 does have a private API. The [./src/core.ts](#) file contain this line:

```
export * from './core_render3_private_export';
```

The [./src/core_render3_private_export.ts](#) file has this:

```
export {
  defineComponent as edefineComponent,
  detectChanges as edetectChanges,
  renderComponent as erenderComponent,
  ComponentType as eComponentType,
  C as eC, E as eE, L as eL, T as eT, V as eV, b as eb, b1 as eb1, b2 as eb2,
  b3 as eb3, b4 as eb4, b5 as eb5, b6 as eb6, b7 as eb7, b8 as eb8, bV as ebV,
  cR as ecR, cr as ecr, e as ee, p as ep, s as es, t as et, v as ev, r as er,
} from './render3/index';
```

Private APIs are intended for use by other Angular packages and not by regular Angular applications. Hence the Greek theta character (' θ ') is added as a prefix to private APIs, as in common with other such private APIs within Angular.

The reason for the many very short type names is that the Angular Compiler will be generating lots of source code based on Render3 for your application's Angular template files and it is desirable to have this as compact as possible, without the need to run a minifier. Typically no human reads this generated code, so compactness is desired rather than readability.

If we examine:

- [<ANGULAR-MASTER>/packages/core/src/Render3/index.ts](#)

we see it starts by explaining the naming scheme:

```
// Naming scheme:
// - Capital letters are for creating things:
// T(Text), E(Element), D(Directive), V(View),
```

```
// C(Container), L(Listener)
// - lower case letters are for binding: b(bind)
// - lower case letters are for binding target:
//     p(property), a(attribute), k(class), s(style), i(input)
// - lower case letters for guarding life cycle hooks: l(lifeCycle)
// - lower case for closing: c(containerEnd), e(elementEnd), v(viewEnd)
```

Then it has a long list of exports of instructions, many with abbreviations:

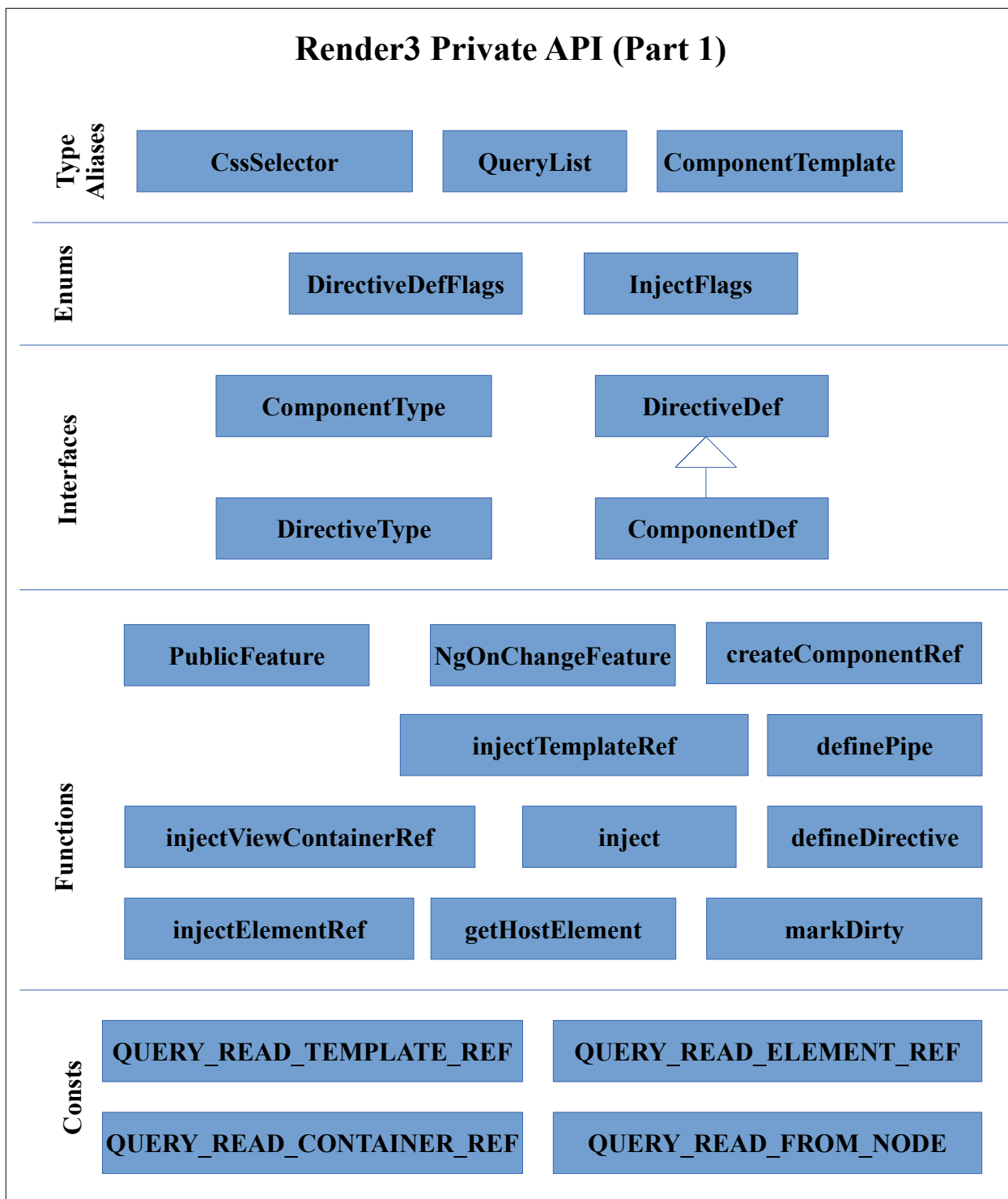
<pre>Export { QUERY_READ_CONTAINER_REF, QUERY_READ_ELEMENT_REF, QUERY_READ_FROM_NODE, QUERY_READ_TEMPLATE_REF, InjectFlags, inject, injectElementRef, injectTemplateRef, injectViewContainerRef } from './di'; export { NO_CHANGE as NC, bind as b, bind1 as b1, bind2 as b2, bind3 as b3, bind4 as b4, bind5 as b5, bind6 as b6, bind7 as b7, bind8 as b8, bindV as bV, componentRefresh as r, container as C, containerRefreshStart as cR, containerRefreshEnd as cr, elementAttribute as a, elementClass as k, elementEnd as e, elementProperty as p, elementStart as E, elementStyle as s, listener as L, memory as m, projection as P, projectionDef as pD, text as T, textBinding as t, embeddedViewStart as V, embeddedViewEnd as v, } from './instructions'; export { pipe as Pp,</pre>	<pre> pipeBind1 as pb1, pipeBind2 as pb2, pipeBind3 as pb3, pipeBind4 as pb4, pipeBindV as pbV, } from './pipe'; export { QueryList, query as Q, queryRefresh as qR, } from './query'; export { objectLiteral1 as o1, objectLiteral2 as o2, objectLiteral3 as o3, objectLiteral4 as o4, objectLiteral5 as o5, objectLiteral6 as o6, objectLiteral7 as o7, objectLiteral8 as o8, } from './object_literal'; export { ComponentDef, ComponentTemplate, ComponentType, DirectiveDef, DirectiveDefFlags, DirectiveType, NgOnChangesFeature, PublicFeature, defineComponent, defineDirective, definePipe, }; export {createComponentRef, detectChanges, getHostElement, markDirty, renderComponent}; export {CssSelector} from './interfaces/projection';</pre>
---	---

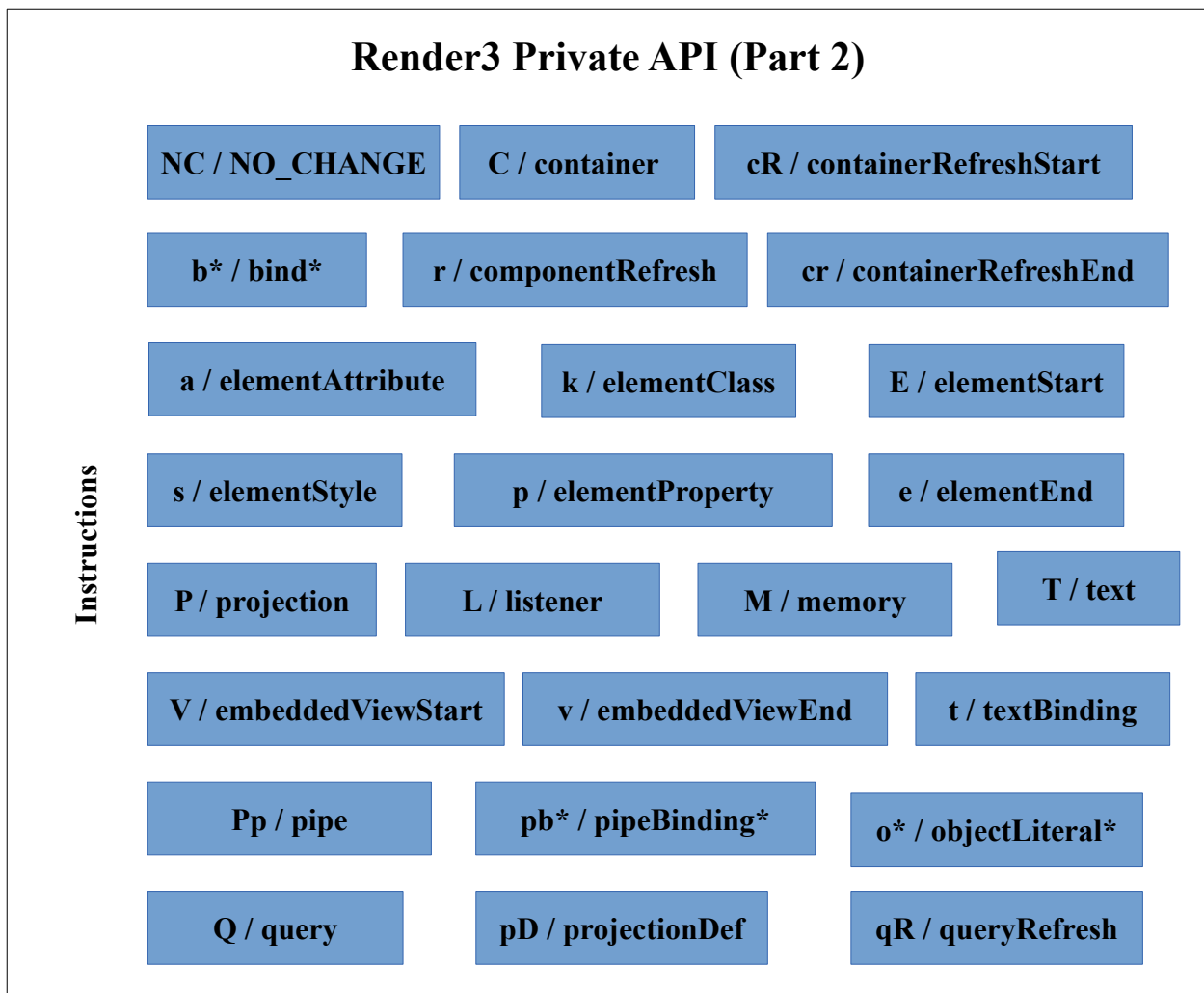
Each of the one- or two-letter exports corresponds to an instruction in [./src/Render3/instructions.ts](#). In the following diagram we give the short export name

and the full name, which more clearly explains the intent of the instruction. We have seen how Core’s Render3 is being used by the compiler and compiler-cli packages:

- [<ANGULAR-MASTER>/packages/compiler/src/render3](#)
- [<ANGULAR-MASTER>/packages/compiler-cli/src/transformers/r3_transform.ts](#)

It is not used by the Router or the platform- packages (which do use Render2).





Source Model

The source tree for the Render3 feature directly contains these source files:

- assert.ts
- component.ts
- definition.ts
- di.ts
- hooks.ts
- index.ts
- instructions.ts
- ng_dev_mode.ts
- node_assert.ts
- node_manipulation.ts
- node_selector_matcher.ts
- object_literal.ts
- pipe.ts
- query.ts
- util.ts

and these documents:

- perf_notes.md

- TREE_SHAKING.md

It also has one sub-directory, `interfaces`, which contains these files:

- `container.ts`
- `definition.ts`
- `injector.ts`
- `node.ts`
- `projection.ts`
- `query.ts`
- `renderer.ts`
- `view.ts`

It could be said Render3 is a re-imagining of what rendering means for an Angular application. The principal change is that the DOM comes back as the main API that is used to render, and the idea of custom renderers goes away. In scenarios where the DOM does not exist (such as a web worker or on the server), then a polyfill DOM will be needed.

In some pieces of Render3 code, we see use of the `'L'` and `'R'` prefixes. This is explained in a comment in the source:

```
The "L" stands for "Logical" to differentiate between `RNodes` (actual rendered DOM node) and our logical representation of DOM nodes, `LNodes`.
<ANGULAR-MASTER>/packages/core/src/render3/interfaces/node.ts
```

Interfaces

When trying to figure out how Render3 works, a good place to start is with its interfaces. Let's look at this first:

- <https://github.com/angular/angular/blob/master/packages/core/src/render3/interfaces/renderer.ts>

There are some simple helper interfaces describing a node, an element and a text node. The node is defined as have three methods to insert, append and remove a child:

```
/** Subset of API needed for appending elements and text nodes. */
export interface RNode {
  removeChild(oldChild: RNode): void;

  // Insert a child node.
  // Used exclusively for adding View root nodes into ViewAnchor location.
  insertBefore(
    newChild: RNode, refChild: RNode|null, isViewRoot: boolean): void;

  //Append a child node.
  //Used exclusively for building up DOM which are static (ie not View roots)
  appendChild(newChild: RNode): RNode;
}
```

The element allows adding and removing of listeners, working with attributes and properties and style configuration – it is defined as:

```
/**
 * Subset of API needed for writing attributes, properties, and setting up
 * listeners on Element.
 */
export interface RElement extends RNode {
```

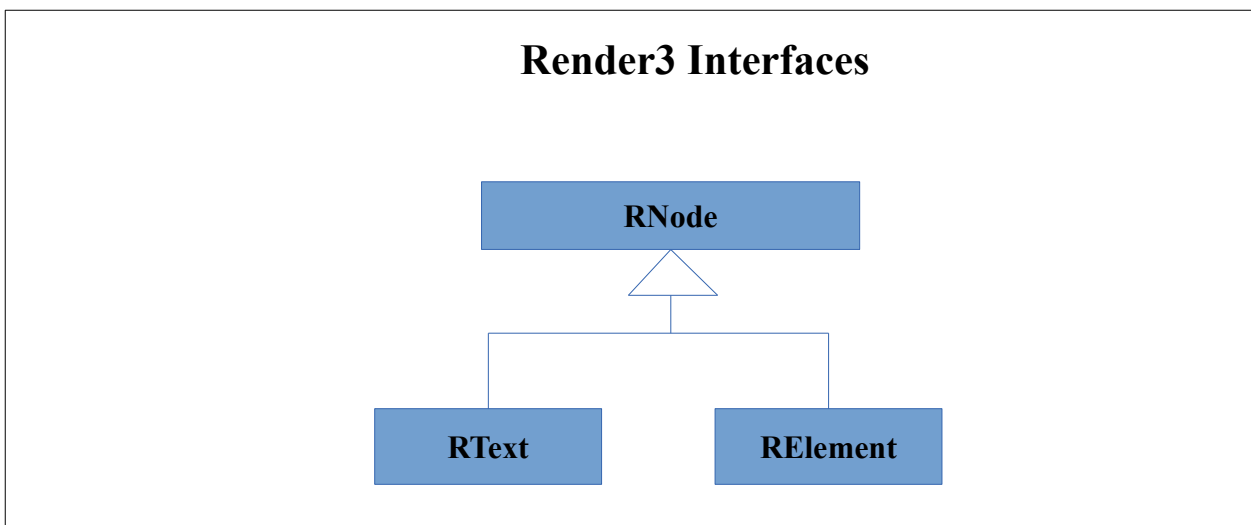
```

style: RCssStyleDeclaration;
classList: RDomTokenList;
setAttribute(name: string, value: string): void;
removeAttribute(name: string): void;
setAttributeNS(
  namespaceURI: string, qualifiedName: string, value: string): void;
addEventListener(
  type: string, listener: EventListener, useCapture?: boolean): void;
removeEventListener(
  type: string, listener?: EventListener, options?: boolean): void;
setProperty?(name: string, value: any): void;
}

```

The text node adds a `textContent` property:

```
export interface RText extends RNode { textContent: string|null; }
```



It has this factory code. Note the return type from `createRenderer` is `Renderer3` **1** – and that for the `domRendererFactory3` implementation **2** this is the normal DOM document:

```

export interface RendererFactory3 {
  createRenderer(hostElement: RElement|null,
    rendererType: RendererType2|null): Renderer3; 1
  begin?(): void;
  end?(): void;
}
export const domRendererFactory3: RendererFactory3 = {
  createRenderer: (
    hostElement: RElement | null,
    rendererType: RendererType2 | null): Renderer3 => { return document; } 2
};

```

This is key to moving back to regular DOM usage for code that runs in the main browser UI thread, and yet allowing alternatives elsewhere. `Renderer3` is a type alias:

```
export type Renderer3 = ObjectOrientedRenderer3 | ProceduralRenderer3;
```

This represents the two kinds of renderers that are supported. The bolded text in the comment highlights the usage scenario for the first of these:

```

/**
 * Object Oriented style of API needed to create elements and text nodes.

```

```

*
* This is the native browser API style, e.g. operations are methods on
* individual objects like HTML<Element>. With this style, no additional
* code is needed as a facade (reducing payload size).
*/
export interface ObjectOrientedRenderer3 {
  createElement(tagName: string): RElement;
  createTextNode(data: string): RText;
  querySelector(selectors: string): RElement|null;
}

```

ProceduralRender3 is intended to be used from web workers and server-side:

```

/**
* Procedural style of API needed to create elements and text nodes.
*
* In non-native browser environments (e.g. platforms such as web-workers),
* this is the facade that enables element manipulation. This also
* facilitates backwards compatibility with Renderer2.
*/
export interface ProceduralRenderer3 {
  destroy(): void;
  createElement(name: string, namespace?: string|null): RElement;
  createText(value: string): RText;
  destroyNode?: ((node: RNode) => void)|null;
  appendChild(parent: RElement, newChild: RNode): void;
  insertBefore(parent: RNode, newChild: RNode, refChild: RNode|null): void;
  removeChild(parent: RElement, oldChild: RNode): void;
  selectRootElement(selectorOrNode: string|any): RElement;
  setAttribute(el: RElement, name: string, value: string,
    namespace?: string|null): void;
  removeAttribute(el: RElement, name: string, namespace?: string|null): void;
  addClass(el: RElement, name: string): void;
  removeClass(el: RElement, name: string): void;
  setStyle(el: RElement, style: string, value: any,
    flags?: RendererStyleFlags2|RendererStyleFlags3): void;
  removeStyle(el: RElement, style: string,
    flags?: RendererStyleFlags2|RendererStyleFlags3): void;
  setProperty(el: RElement, name: string, value: any): void;
  setValue(node: RText, value: string): void;
  listen(target: RNode, eventName: string,
    callback: (event: any) => boolean | void): () => void;
}

```

Let's now look at [view.ts](#). It includes the following to work with static data:

```

// The static data for an LView (shared between all templates of a
// given type). Stored on the template function as ngPrivateData.
export interface TView {
  data: TData;
  firstTemplatePass: boolean;
  initHooks: HookData|null;
  checkHooks: HookData|null;
  contentHooks: HookData|null;
  contentCheckHooks: HookData|null;
  viewHooks: HookData|null;
  viewCheckHooks: HookData|null;
  destroyHooks: HookData|null;
  objectLiterals: any[]|null;
}

```

TData is defined as:

```
/**
 * Static data that corresponds to the instance-specific data array on an
 * LView. Each node's static data is stored in tData at the same index that
 * it's stored in the data array. Each directive's definition is stored here
 * at the same index as its directive instance in the data array. Any nodes
 * that do not have static data store a null value in tData to avoid a
 * sparse array.
 */
export type TData = (TNode | DirectiveDef<any>| null) [];
```

A tree of LViews or LContainers will be needed, so this type is a node in the hierarchy:

```
/** Interface necessary to work with view tree traversal */
export interface LViewOrLContainer {
  next: LView|LContainer|null;
  child?: LView|LContainer|null;
  views?: LViewNode[];
  parent: LView|null;
}
```

An LView stores info relating to processing a view's instructions. Detailed comments (not shown here) for each of its fields are in [the source](#).

```
/**
 * `LView` stores all of the information needed to process the instructions
 * as they are invoked from the template. Each embedded view and component
 * view has its own `LView`. When processing a particular view, we set the
 * `currentView` to that `LView`. When that view is done processing, the
 * `currentView` is set back to whatever the original `currentView` was
 * before (the parent `LView`).
 * Keeping separate state for each view facilities view insertion / deletion,
 * so we don't have to edit the data array based on which views are present.
 */
export interface LView {
  creationMode: boolean;
  readonly parent: LView|null;
  readonly node: LViewNode|LElementNode;
  readonly id: number;
  readonly renderer: Renderer3;
  bindingStartIndex: number|null;
  cleanup: any[]|null;
  lifecycleStage: LifecycleStage;
  child: LView|LContainer|null;
  tail: LView|LContainer|null;
  next: LView|LContainer|null;
  readonly data: any[];
  tView: TView;
  template: ComponentTemplate<{}>|null;
  context: {}|null;
  dynamicViewCount: number;
  queries: LQueries|null;
}
```

The [container.ts](#) file looks at containers, which are collections of views and sub-containers. It exports one type, TContainer:

```
/**
```



```

* The static equivalent of LContainer, used in TContainerNode.
*
* The container needs to store static data for each of its embedded views
* (TViews). Otherwise, nodes in embedded views with the same index as nodes
* in their parent views will overwrite each other, as they are in
* the same template.
*
* Each index in this array corresponds to the static data for a certain
* view. So if you had V(0) and V(1) in a container, you might have:
*
* [
*   [{tagName: 'div', attrs: ...}], null],      // V(0) TView
*   [{tagName: 'button', attrs ...}], null]    // V(1) TView
* ]
*/
export type TContainer = TView[];

```

and one interface, `LContainer`: (note [the source file](#) contained detailed comments for each field):

```

/** The state associated with an LContainer */
export interface LContainer {
  nextIndex: number;
  next: LView|LContainer|null;
  parent: LView|null;
  readonly views: LViewNode[];
  renderParent: LElementNode|null;
  readonly template: ComponentTemplate<any>|null;
  dynamicViewCount: number;
  queries: LQueries|null;
}

```

The [query.ts](#) file contains the `QueryReadType` class:

```

export class QueryReadType<T> { private defeatStructuralTyping: any; }

```

and the `LQuery` interface:

```

/** Used for tracking queries (e.g. ViewChild, ContentChild). */
export interface LQueries {
  child(): LQueries|null;
  addNode(node: LNode): void;
  container(): LQueries|null;
  enterView(newViewIndex: number): LQueries|null;
  removeView(removeIndex: number): void;
  track<T>(
    queryList: QueryList<T>,
    predicate: Type<any>|string[],
    descend?: boolean,
    read?: QueryReadType<T>|Type<T>): void;
}

```

The [projection.ts](#) file define `LProjection` as:

```

// Linked list of projected nodes (using the pNextOrParent property).
export interface LProjection {
  head: LElementNode|LTextNode|LContainerNode|null;
  tail: LElementNode|LTextNode|LContainerNode|null;
}

```

The `injector.ts` file has this:

```

export interface LInjector {
  readonly parent: LInjector|null;
  readonly node: LElementNode|LContainerNode;
  bf0: number;
  bf1: number;
  bf2: number;
  bf3: number;
  cbf0: number;
  cbf1: number;
  cbf2: number;
  cbf3: number;
  injector: Injector|null;
  /** Stores the TemplateRef so subsequent injections of the TemplateRef get
  the same instance. */
  templateRef: TemplateRef<any>|null;
  /** Stores the ViewContainerRef so subsequent injections of the
  ViewContainerRef get the same
  * instance. */
  viewContainerRef: ViewContainerRef|null;
  /** Stores the ElementRef so subsequent injections of the ElementRef get
  the same instance. */
  elementRef: ElementRef|null;
}

```

The `node.ts` file is large and contains a hierarchy of node-related types.

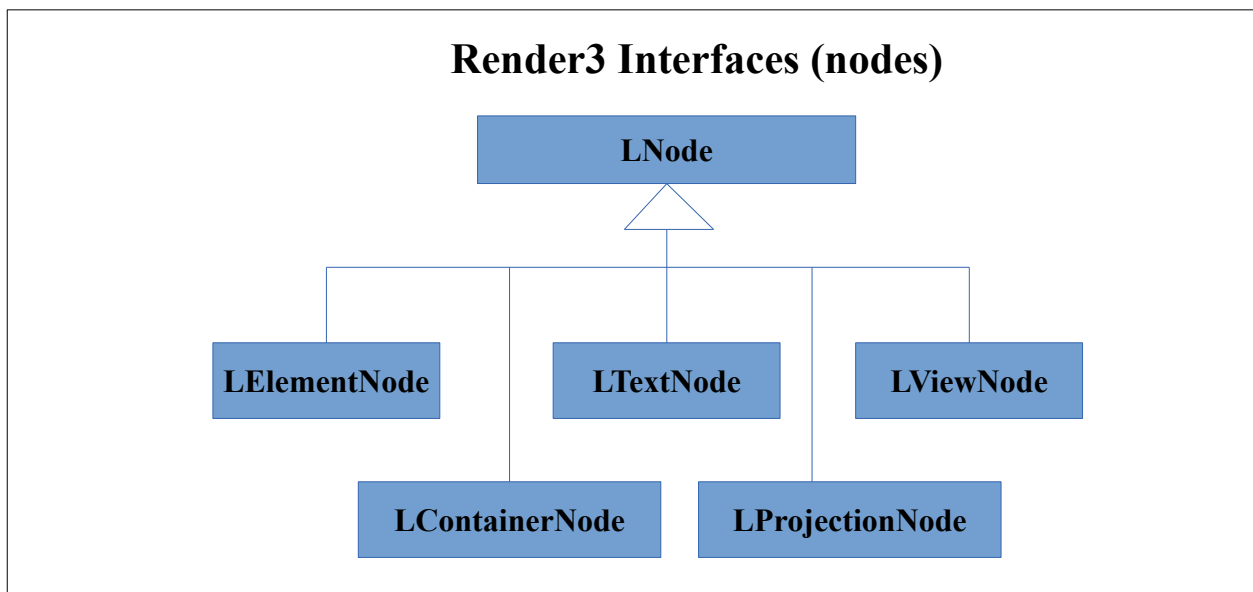
The root, `LNode`, is defined (abbreviated) as:

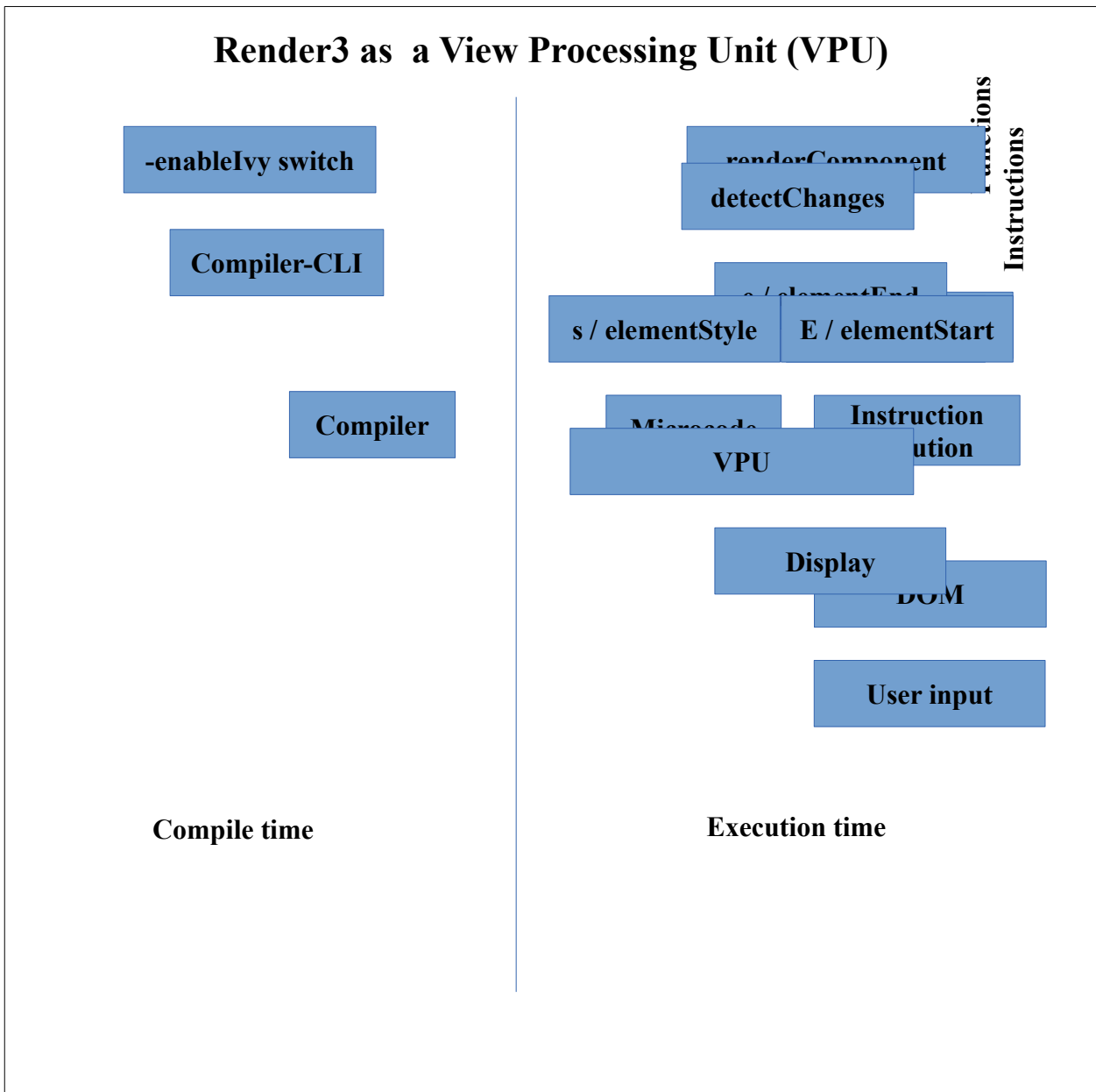
```

/**
 * LNode is an internal data structure which is used for the incremental DOM
 * algorithm. The "L" stands for "Logical" to differentiate between `RNodes`
 * (actual rendered DOM node) and our logical representation of DOM nodes,
 * `Lnodes`. The data structure is optimized for speed and size.
 *
 * In order to be fast, all subtypes of `LNode` should have the same shape.
 * Because size of the `LNode` matters, many fields have multiple roles
 * depending on the `LNode` subtype.
 */
export interface LNode {
  flags: LNodeFlags;
  readonly native: RElement|RText|null|undefined;
  readonly parent: LNode|null;
  child: LNode|null;
  next: LNode|null;
  readonly data: LView|LContainer|LProjection|null;
  readonly view: LView;
  nodeInjector: LInjector|null;
  queries: LQueries|null;
  pNextOrParent: LNode|null;
  tNode: TNode|null;
}

```

Each of the other types adds a few additional fields to represent that node type.





15: The Forms Package

Overview

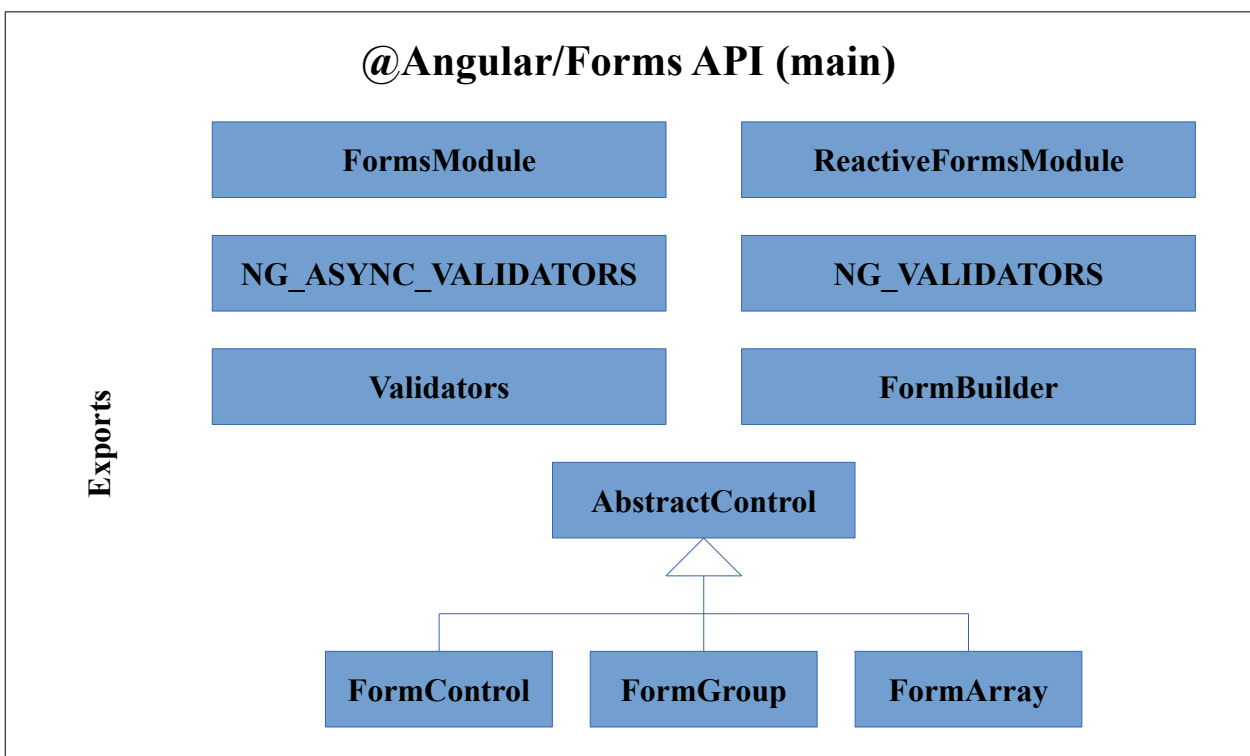
The Forms module provides capabilities to manage web forms.

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';
```

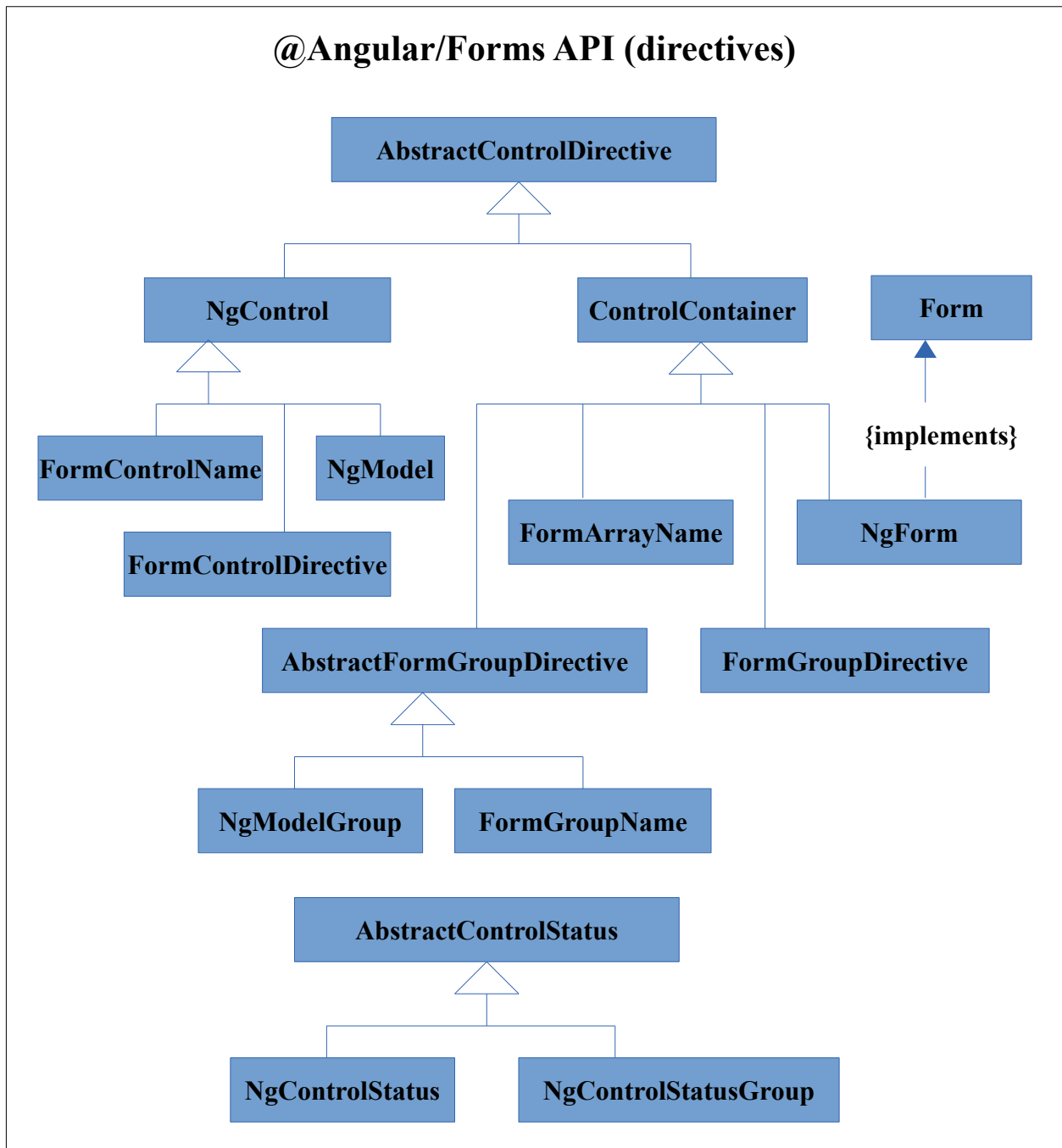
The parts of forms.ts related to exporting the above are:

```
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}

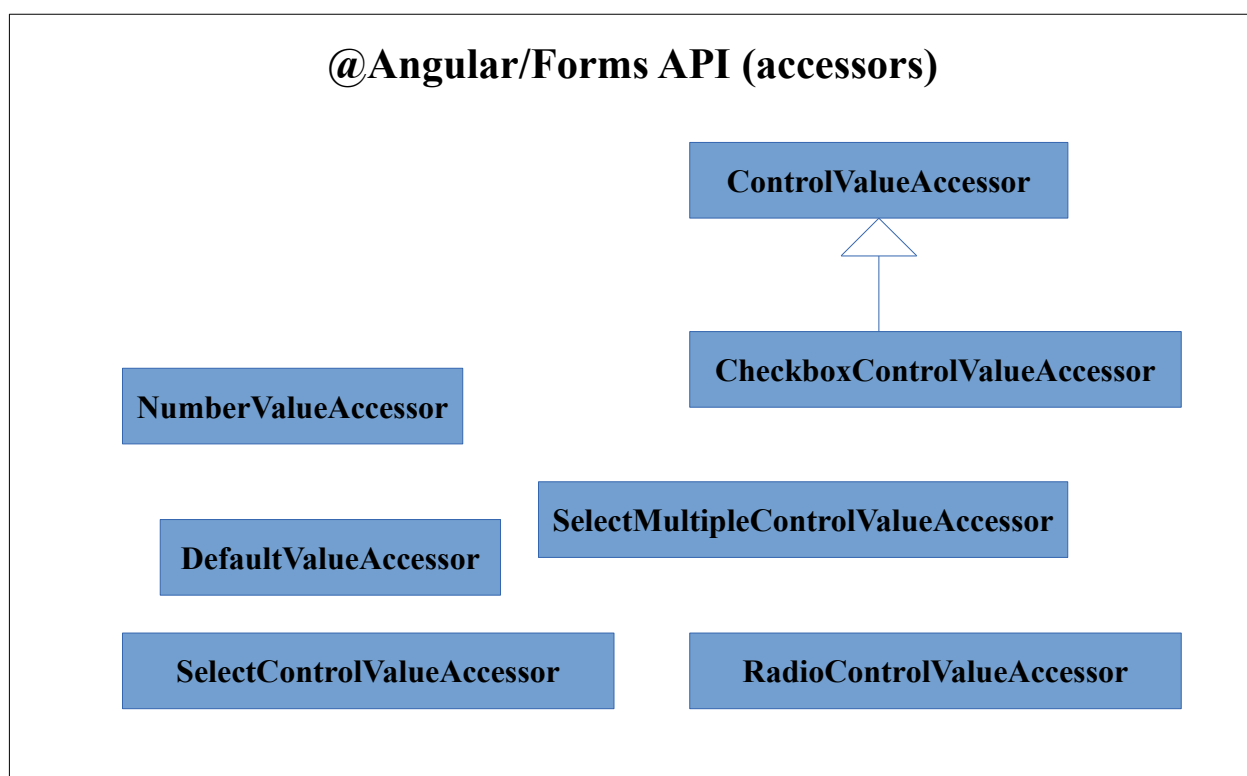
```

```

    from './directives/abstract_form_group_directive';
export {ControlContainer} from './directives/control_container';
export {Form} from './directives/form_interface';
export {FormControl} from './directives/ng_control';
export {FormControlStatus, FormControlStatusGroup}
    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:

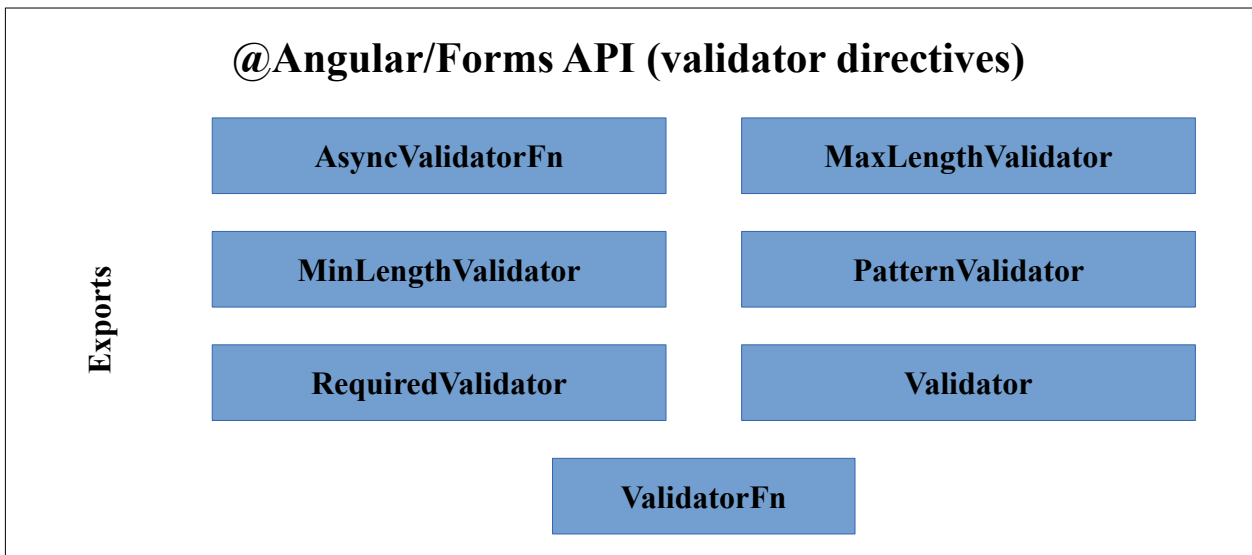
```

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 `FormsModule` 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 `FormBuilder` 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:

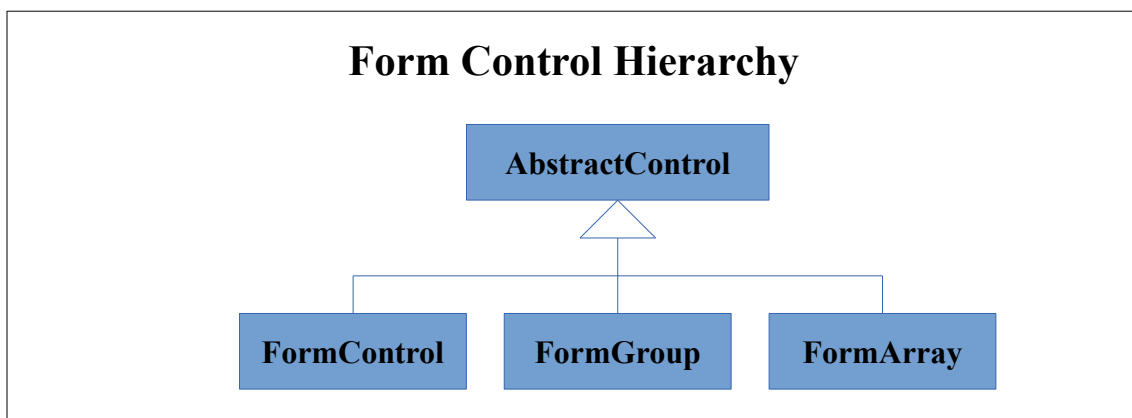
```

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 store 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 `` 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 contrast 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();
  }
```