# Angular CLI

## Angular CLI

Angular now comes with a command line interface (CLI) to make it easier and faster to build Angular applications.

The Angular CLI at the time of writing (09/2016) is still in a prototype stage and has a number of issues that still need to be addressed. The biggest is that it doesn’t currently work with the Angular router.

Even without support for the router however the CLI is a useful if not essential tool.

**Features**

The Angular CLI helps with:

**Bootstrapping a project**

It creates the initial project structure with a root NgModule and a root component and bootstraps it using the platformBootstrapDynamic method.

The project is also configured to use the *webpack loader* which handles things like module loading, bundling and minification of dependant code.

In the course we’ve used SystemJS for this since webpack doesn’t work with

Plunker *yet*. We’ll continue to use SystemJS for the code samples in Plunker and WebPack for any applications created with the Angular CLI.

**Serving and live reloading**

The CLI starts a local web-server so we can view our application in the browser via localhost:4000.

The CLI also watches for any changes to our files and automatically reloads the webpage if there are any.

**Code generation**

Using the CLI we can create components directives, services, pipes etc… all from the command line with all the necessary files, folders and boilerplate code included.

All the generated code adheres to the official Angular [style guide](https://angular.io/styleguide).

In Angular 1 the Angular team never supported an official style guide. This

meant that most projects ended up looked pretty different to each other. Adeveloper moving teams would have to figure out from scratch how *this* team

likes to write Angular 1 code.

**Testing**

The generated code also comes with bootstrapped jasmine test spec files, we can use the CLI to compile and run all the tests with a single command.

Whenever the CLI detects changes to any file it re-runs all the tests automatically in the background.

**Packaging and releasing**

The CLI doesn’t just stop with development, using it we can also package our application ready for release to a server.

**Installing the Angular CLI**

To install the CLI we use Node and npm.

npm install -g angular-cli

If the above ran successfully it will have made available to you a new application called ng, to test this installed correctly run this command:

ng -v

It *should* output the version of the application that was installed, like so:

angular-cli: 1.0.0-beta.15

node: 6.4.0

os: darwin x64

**Start an application with** ng new

Lets create a new project called codecraft.

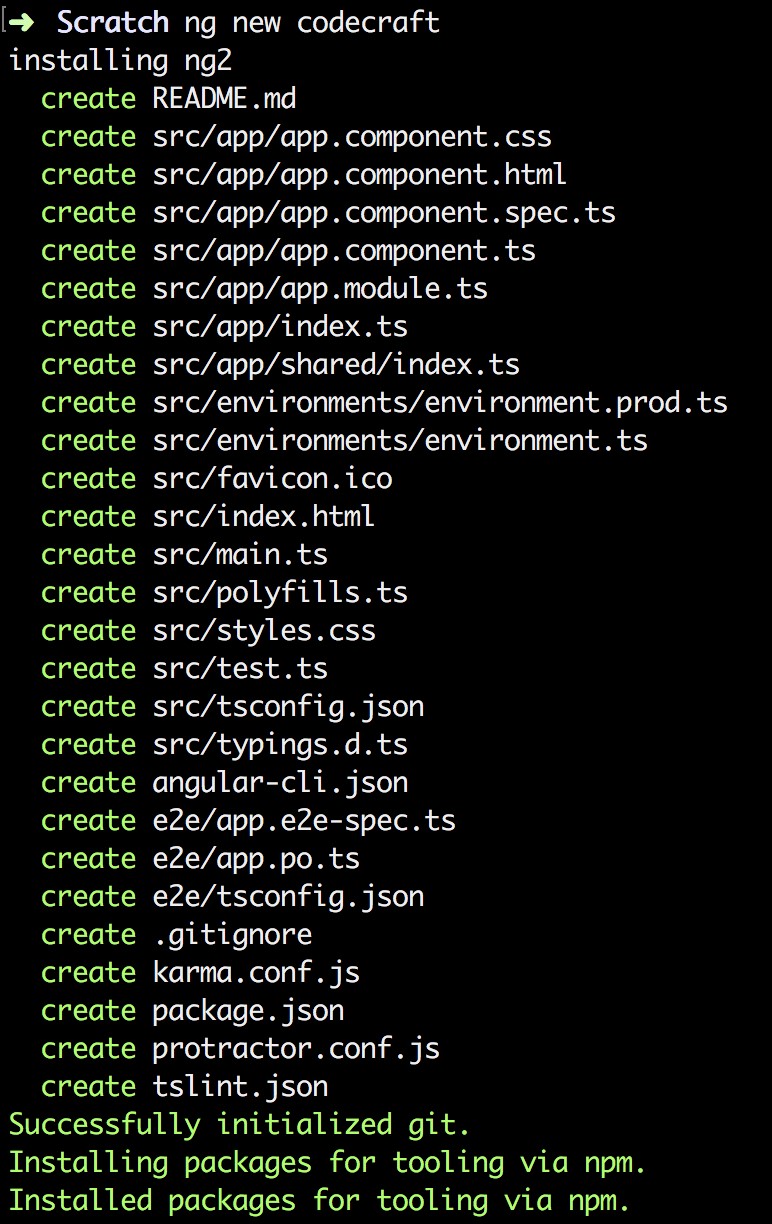
To bootstrap our new project with ng we run this command:

ng new codecraft



This command might take sometime to run, be patient.

This outputs something like the below:



The command generates a number of new files and folders for us:

codecraft

// production or development builds of our applicaiton go here.

├──

dist

// main application code goes here.

├──

src

│

├──

app

│

│

├──

app.component.css

│

│

├──

app.component.html

│

│

├──

app.component.spec.ts

│

│

├──

app.component.ts

│

│

├──

app.module.ts

│

│

├──

index.ts

│

│

└──

shared

│

│

└──

index.ts

// settings for the different environments, dev, qa, prod.

│

├──

environments

│

│

├──

environment.prod.ts

│

│

└──

environment.ts

// main html and typescript file

│

├──

index.html

│

├──

main.ts

│

├──

favicon.ico

│

├──

polyfills.ts

│

├──

styles.css

// prepares test environment and runs all the unit tests

│

├──

test.ts

// typescript configuration file

│

├──

tsconfig.json

// typescript type definition file

│

└──

typings.d.ts

// The E2E tests for our application go here

├──

e2e

├──

angular-cli.json

├──

karma.conf.js

├──

package.json

├──

protractor.conf.js

├──

README.md

└──

tslint.json

 The directory structure follows the **recommended** app structure and style guide.

As well as creating the files and folders for us; we can see from package.json that it installed the correct versions of all the required npm dependencies for us also.

{

"name": "activity",

"version": "0.0.0",

"license": "MIT",

"scripts": {

"ng": "ng",

"start": "ng serve",

"build": "ng build",

"test": "ng test",

"lint": "ng lint",

"e2e": "ng e2e"

},

"private": true,

"dependencies": {

"@angular/animations": "^5.0.0",

"@angular/common": "^5.0.0",

"@angular/compiler": "^5.0.0",

"@angular/core": "^5.0.0",

"@angular/forms": "^5.0.0",

"@angular/http": "^5.0.0",

"@angular/platform-browser": "^5.0.0",

"@angular/platform-browser-dynamic": "^5.0.0",

"@angular/router": "^5.0.0",

"core-js": "^2.4.1",

"rxjs": "^5.5.2",

"zone.js": "^0.8.14"

},

"devDependencies": {

"@angular/cli": "1.5.0",

"@angular/compiler-cli": "^5.0.0",

"@angular/language-service": "^5.0.0",

"@types/jasmine": "~2.5.53",

"@types/jasminewd2": "~2.0.2",

"@types/node": "~6.0.60",

"codelyzer": "~3.2.0",

"jasmine-core": "~2.6.2",

"jasmine-spec-reporter": "~4.1.0",

"karma": "~1.7.0",

"karma-chrome-launcher": "~2.1.1",

"karma-cli": "~1.0.1",

"karma-coverage-istanbul-reporter": "^1.2.1",

"karma-jasmine": "~1.1.0",

"karma-jasmine-html-reporter": "^0.2.2",

"protractor": "~5.1.2",

"ts-node": "~3.2.0",

"tslint": "~5.7.0",

"typescript": "~2.4.2"

}

}

So far in this course we have bundled all our code into one file on plunker for convenience.

Lets see how the Angular CLI breaks up the code into multiple files and where those files are located.

**src/app/app.component.ts**

The new project is bootstrapped with one component, our root component which it called AppComponent and has a selector of app-root.

*src/app/app.component.ts*

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

}

**src/index.html** app-root component has been added to our index.html file already.

There are no script tags present yet, that’s fine the angular build process adds all the required script and link tags for us.

*src/index.html*

<!

doctype html

>

<

html lang="en"

>

<

head

>

<meta charset="utf-8">

<title>Activity</title>

<base href="/">

<link rel="stylesheet"

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-

alpha.4/css/bootstrap.min.css">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

<

/head

>

<

body

>

<app-root></app-root>

<

/body

>

<

/html

>

**src/app/app.module.ts**

Our top level module configuration is stored in this file.

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

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';

import { JokeComponent } from './joke/joke.component';

import { JokeListComponent } from './joke-list/joke-list.component';

import { JokeFormComponent } from './joke-form/joke-form.component';

import { HeaderComponent } from './header/header.component';

@NgModule({

declarations: [

AppComponent,

JokeComponent,

JokeListComponent,

JokeFormComponent,

HeaderComponent

],

imports: [

BrowserModule,

FormsModule,

HttpModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**src/main.ts**

The actual act of importing our main module and boostrapping our Angular web application is left to the main.ts file.

import { enableProdMode } from '@angular/core';

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

import { AppModule } from './app/app.module'; import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(err => console.log(err));

**Serve an application with** ng serve

With the CLI we can also easily serve our application using a local web-server.

We just run:

ng serve

This builds our application, bundles all our code using webpack and makes it all available through localhost:4200.

ng serve also watches for any changes to files in our project and auto-reloads the browser for us.

The command runs the application through a web-server that support HTML5 *push-state* routing.  The above will make sense once we cover Routing later on in this book.

**Generate code with** ng generate

The ability to generate stub code is one of the most useful features of the CLI.

The most exciting part of this is that it automatically generates code that adheres to the official style guide. Projects built using the Angular CLI should look like each other. Developers who

are used to the way Angular CLI generates files are going to find it easier to work on multiple different projects, as long as they all use the Angular CLI.

With the generate command we can create new components, directives, ~~routes~~ *not available in version 1.0.0-beta.15*, pipes, services, classes, interfaces and enums.

Each of the above *types of things* it can create is called a **scaffold**.

We can run this command using ng generate <scaffold> <name>

If we wanted to generate a component called HeaderComponent we would write:

ng generate component Header

This creates a number of files in a folder called header in src/app, like so:

app

├── header

│ ├── header.component.css // The css for this component

│ ├── header.component.html // The template for this component

│ ├── header.component.spec.ts // The unit test for this component │ └── header.component.ts // The component typescript file

Taking a look at

header.component.ts

:

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-header',

templateUrl: './header.component.html',

styleUrls: ['./header.component.css']

})

export class HeaderComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}



Don’t name your component

HeaderComponent

. Angular CLI automatically appends

Component

to the name, so you component class would end up being

HeaderComponentComponent

.

The command above can be shortened to:

ng g c Header

If we run the command in an app folder, the generate command will create files **relative to the current folder you are in**. So if we are in src/app/header and we run ng g c LoginButton it will generate the files in src/app/header/login-button/



We can also be explicit about where we want the generated files to go by running ng g component ./src/app/foo/bar this will create a component called BarComponent in the folder ./src/app/foo/bar.

### Available Scaffolds

**Component**

ng g component My // Creates MyComponent

By default all generated files go in into src\app\my-component, a folder called my-component is created for us.

**Directive**

ng g directive My // Creates MyDirective

By default all generated files go in into src\app, *no folder is created*.

**Pipe**

ng g pipe My // Creates MyPipe

By default all generated files go in into src\app, *no folder is created*.

**Service**

ng g service MyService // Creates MyService

By default all generated files go in into src\app, *no folder is created*.

**Class**

ng g class MyClass // Creates MyClass

By default all generated files go in into src\app, *no folder is created*.

**Interface**

ng g interface MyInterface // Creates MyInterface

By default all generated files go in into src\app, *no folder is created*.

**Enum**

ng g enum MyEnum // Creates MyEnum

By default all generated files go in into src\app, *no folder is created*.

**Create a build with** ng build

The ng serve command does a great job of enabling development locally.

However *eventually* we will want some code which we can host on another server somewhere.

The Angular CLI again has us covered in this regard, if we want to create a development build we simply type

ng build

This bundles all our javascript, css, html into a smaller set of files which we can host on another site

simply.

It outputs these files into the

dist

folder:

.

├──

assets

├──

index.html

├──

inline.js

├──

inline.map

├──

main.bundle.js

├──

main.map

├──

styles.bundle.js

└──

styles.map

To serve our built application site we just need to serve this folder. For example if using python we could simply run python -m SimpleHTTPServer from the dist folder and view the application from 0.0.0.0:8000.

### Production Builds

By default the ng build command creates a development build, no effort is made to optimise the code.

To create a production build we just run

ng build --prod

This might generate an output like the below:

.

├──

assets

├──

index.html

├──

inline.js

├──

main.3f26904b701596b6d90a.bundle.js

├──

main.3f26904b701596b6d90a.bundle.js.gz

└──

styles.b52d2076048963e7cbfd.bundle.js

Running with --prod changes a few things:

* The bundles now have random strings appended to them to enable **cache busting**.

This ensures that a browser doesn’t try to load up previously cached versions of the files and instead load the new ones from the server.

* The file sizes are much smaller. The files have been processed through a minifier and uglifier.
* There is a much small .gz file, this is a compressed version of the equivalent javascript file. Browsers will automatically try to download the .gz version of files if they are present.

### Adding a third party module

The build system simplifies the process of serving and releasing your application considerably. It works only because Angular knows about all the files used by your application.

So when we include 3rd party libraries into our application we need to do so in such a way that Angular knows about the libraries and includes them in the build process.

**Bundled with the main application javascript files**

If we want to include a module to use in our Angular javascript code, perhaps we want to to use the moment.js library, we just need to install it via npm like so:

npm install moment --save

If we also want to include the typescript type definition file for our module we can install it via:

npm install @types/moment --save

Now when Angular create a build either when releasing or serving locally, the moment library is automatically added to the bundle.

**Global Library Installation**

Some javascript libraries need to be added to the global scope, and loaded as if they were in a script tag.

We can do this by editing the angular-cli.json file in our project root.

The twitter bootstrap library is a great example of this, we need to incldue css and script files in the global scope.

First we install the bootstrap library via npm like so:

npm install bootstrap@next

Then we add the required javascript files to the app.scripts section or the app.styles in angularcli.json like so:

{

.

.

.

"apps": [

{

.

.

.

"styles": [

"styles.css",

"../node\_modules/bootstrap/dist/css/bootstrap.css"

],

"scripts": [

"../node\_modules/jquery/dist/jquery.js",

"../node\_modules/tether/dist/js/tether.js",

"../node\_modules/bootstrap/dist/js/bootstrap.js"

],

.

.

.

}

],

.

.

.

}

Now when the build runs the CLI includes those files in the bundle and injects then in the global scope.

**Testing Angular**

Angular has always been synonymous with testing and so there should be no surprise that the command line tool comes with features to make Angular testing easier.

The default mechanism for unit testing in Angular is via jasmine and karma.

Whenever we generate code via scaffolds it also generates a .spec.ts. The code the CLI bootstraps inside this file depends on the scaffold type but essentially is a jasmine test spec which you can flesh out with more test cases.

 These types of tests are called we only test one *unit* of code and each test case is independent of the others.*unit tests* because we should be writing the tests so

We can run all our unit tests with one command:

ng test

This builds our project and then runs all the tests, any errors are output to the terminal.

This command also watches for any changes in our files and, if it detects any, re-runs the tests automatically.





When running the tests it opens up a browser window like the example above.

It needs this browser windows to run the tests, do not close it!

**Summary**

The above is just an overview of the main commands and their default features.

To find out more details about each command and how we can customise the behaviour via flags we can run ng help in the terminal.

By handling the setup for us the CLI has made working with Angular much easier.

By standardising setup and structure it’s also made Angular projects fungible. Angular developers used to the Angular CLI should feel comfortable on **all** Angular CLI projects and be able to hit the floor running.

## Activity

In this activity you will re-create the Joke application we’ve ran using Plunker as an Angular CLI project instead.

**Steps**

1. Ensure you have *node* installed on your computer. If you are unsure how to install node then follow the instructions in the Node lecture in the Appendix to this course.
2. Follow the instructions in the previous lectures and on the Angular CLI repository [site](https://github.com/angular/angular-cli) to install the Angular CLI on your computer.
3. Crete a project locally on your computer.
4. Mirror the functionality of the Joke application in the plunker below in your Angular CLI powered project.

<http://plnkr.co/edit/b0F6Dhb40Hm5zfiamAix?p=preview>

**Solution**

When you are ready compare your answer to the solution in this repository: [https://github.com/](https://github.com/codecraftpro/angular-2-book-cli-activity)