

InfoWorld DeepDive

FROM IDG

***GET
STARTED
WITH***

TUTORIAL

Angular



Deep Dive

**ANGULAR
TUTORIAL**

GET STARTED WITH Angular

*A step-by-step
guide to installing
the tools, creating
an application,
and getting up to
speed with Angular
components,
directives, services,
and routers*

BY MARTIN HELLER

Angular, the successor to AngularJS, is a development platform for building mobile and desktop applications using TypeScript and/or JavaScript and other languages. Angular is popular for building high-volume websites and it supports web, mobile web, native mobile, and native desktop applications.

The Angular core development team is split between Google employees and a robust community; it's not going away any time soon. In addition to its own extensive capabilities, the Angular platform has a strong external ecosystem: Several prominent IDEs support Angular, it has four data libraries, there are half a dozen useful tools and over a dozen sets of UI components, and there are dozens of Angular books and courses.

[In 2015 when I awarded AngularJS a BOSSie,](#) I explained that it is a model-view-whatever (MVW) JavaScript AJAX framework that extends HTML with markup for dynamic views and two-way data binding. Angular is especially good for developing single-page web applications and linking HTML forms to models and JavaScript controllers. The new Angular is written in TypeScript rather than JavaScript, which has many benefits, as I'll explain.



Deep Dive



In general Angular has good tooling and is suitable for really large, high-traffic projects.

The weird-sounding “model-view-whatever” pattern is an attempt to include the model-view-controller (MVC), model-view-view-model (MVVM), and model-view-presenter (MVP) patterns under one moniker. The differences between these three closely related patterns are the sorts of things that programmers love to argue about fiercely; the Angular developers decided to opt out of the discussion.

Basically, Angular automatically synchronizes data from your UI (views in AngularJS and templates in Angular 2 and above) with your JavaScript objects (model) through two-way data binding. To help you structure your application better and make it easy to test, Angular teaches the browser how to do dependency injection and inversion of control. The new Angular (version 2 and above) replaces views and controllers with components and adopts standard conventions, which makes it easier to understand, and allows developers to concentrate on developing ECMAScript 6 modules and classes. In other words, Angular 2 is a total rewrite of AngularJS that tries to implement the same ideas in a better way.

Angular view templates, which have a fairly simple syntax, are compiled into JavaScript that is well optimized for modern JavaScript engines. The new component router in Angular 2 can do code-splitting (lazy loading) to reduce the amount of code delivered to render a view.

Why Angular? And when is it not a good choice?

Choosing a JavaScript framework for a web app is the sort of process that sets off religious wars among developers. I’m not here to proselytize Angular, but I do want you to know its advantages and disadvantages. Ideally, you should pick the framework that’s appropriate for your app, taking into account the skills in your organization and the frameworks you are using in other applications.

In general Angular has good tooling and is suitable for really large, high-traffic projects. Angular, as a complete rewrite from AngularJS, was designed from the ground up for use on mobile devices and for high performance. Like its predecessor, it does data binding easily and well.

Angular uses a web component pattern, but not Web Components per se. It’s not Polymer, which creates real Web Components, although you can use Polymer Web Components in Angular applications if you wish. Angular does use inversion of control (IoC) and dependency injection (DI) patterns, and fixes some problems with the AngularJS implementation of these.

Angular uses directives and components that mix logic with HTML markup. One school of thought says that mixing logic with presentation is a cardinal sin. Another school of thought says that having everything a program does declared in one place makes it easier to develop and understand. That’s an issue you’ll have to decide for yourself, as I’ve found myself on different sides of the question for different projects.



Deep Dive

Angular does have some documentation issues, frequent backward-compatibility problems, and many concepts for a new developer to learn. On the other hand, Angular has a huge ecosystem that fills the gaps in Angular's documentation with third-party web tutorials, videos, and books.

About TypeScript

Angular is implemented in TypeScript, a duck-typed superset of JavaScript that transpiles to JavaScript. In general, TypeScript applications are easier to maintain at production scale than JavaScript. The simple process of determining whether your types are correct at compile time eliminates a large class of common JavaScript errors, and knowing the types allows editors, tools, and IDEs to be more helpful with code completion, refactoring, and code checking.

I happen to be a big fan of TypeScript. I find it to be much more productive to work on a large TypeScript project than to work on a large JavaScript project. With JavaScript, I really never know whether bugs are lurking in the code waiting to bite me, no matter how often I run JSHint. With TypeScript, at least when I've added the optional types, classes, modules, and interfaces, I feel much more secure.

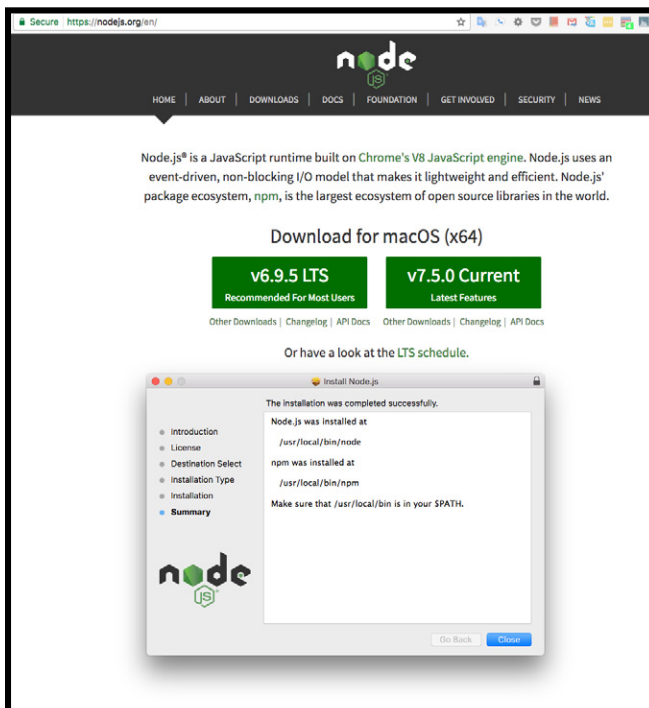
Get started: Install Angular, TypeScript, and Visual Studio Code

With that said, let's install the software and get started.

Install Node.js and NPM

Before we do anything else, we need to install Node.js and NPM, the Node package manager, because they underlie much of Angular's installation and tooling. To find out whether they are installed, and if so, which versions are installed, go to a console or terminal prompt and type the following two commands:

```
$ node -v
$ npm -v
```



On my machine, the Node.js version reported is v6.9.5 and the NPM version is 3.10.10.

Those are the current long-term-support versions at the moment, as I can tell from looking at <https://nodejs.org/>.

If your versions are current, you can skip to the next section. If either command is not found or either version is out of date, you should install the current versions. My versions are current because I recently reinstalled Node, as shown in the screenshot at left.

Installing both Node.js and NPM is a matter of browsing to nodejs.org, pressing the green LTS button, and following the instructions. Once you've completed the installation, check the versions again to make sure they really updated. I know repeating the check sounds paranoid, but a good programmer needs a healthy dose of paranoia to avoid bugs, and aborted installations aren't uncommon.

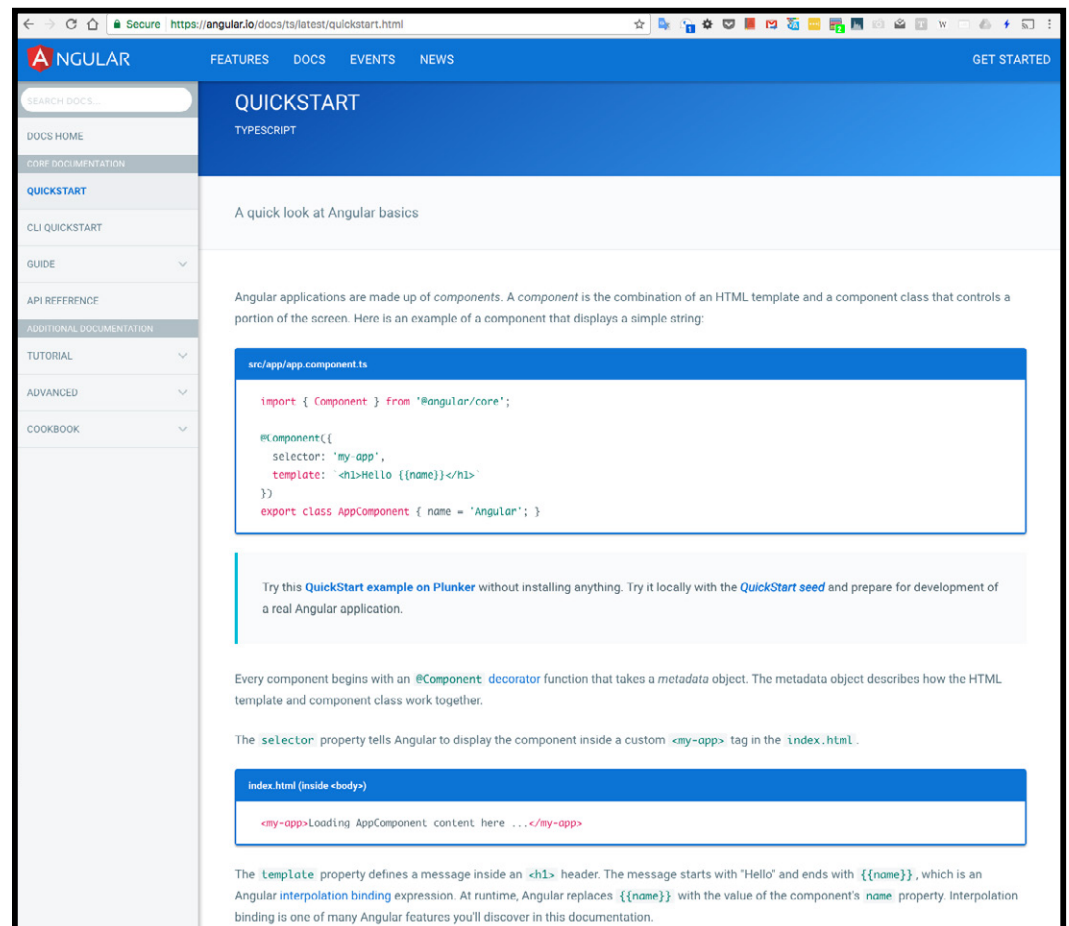
1. Install Angular

There are two ways to install and use Angular. I'll show you the command-line interface (CLI) method first, for several reasons. The first is that it fits better with the way I like to work. The second is that the CLI generates a more complete starter application than the QuickStart seed. The third is that the cleanup step

Deep Dive

in the QuickStart seed instructions seems like it might wreak havoc if used at the wrong time or in the wrong directory.

Browse to <https://angular.io/> and click on one of the three “get started” buttons. They all go to the same place, the Angular QuickStart:



Please read that page over, and feel free to try the QuickStart example on Plunker via the link after the first code block. Once you think you can follow the **@Component** decorator function and the Angular interpolation binding expression **{{name}}**, click on the **CLI QuickStart** link at the left. Don't worry too much about how the decorator function and interpolation binding are implemented: We'll get to that.

1a. Install and test Angular-CLI

We're going to follow the instructions to set up the CLI development environment. The first step is to install Angular and its CLI globally with npm:

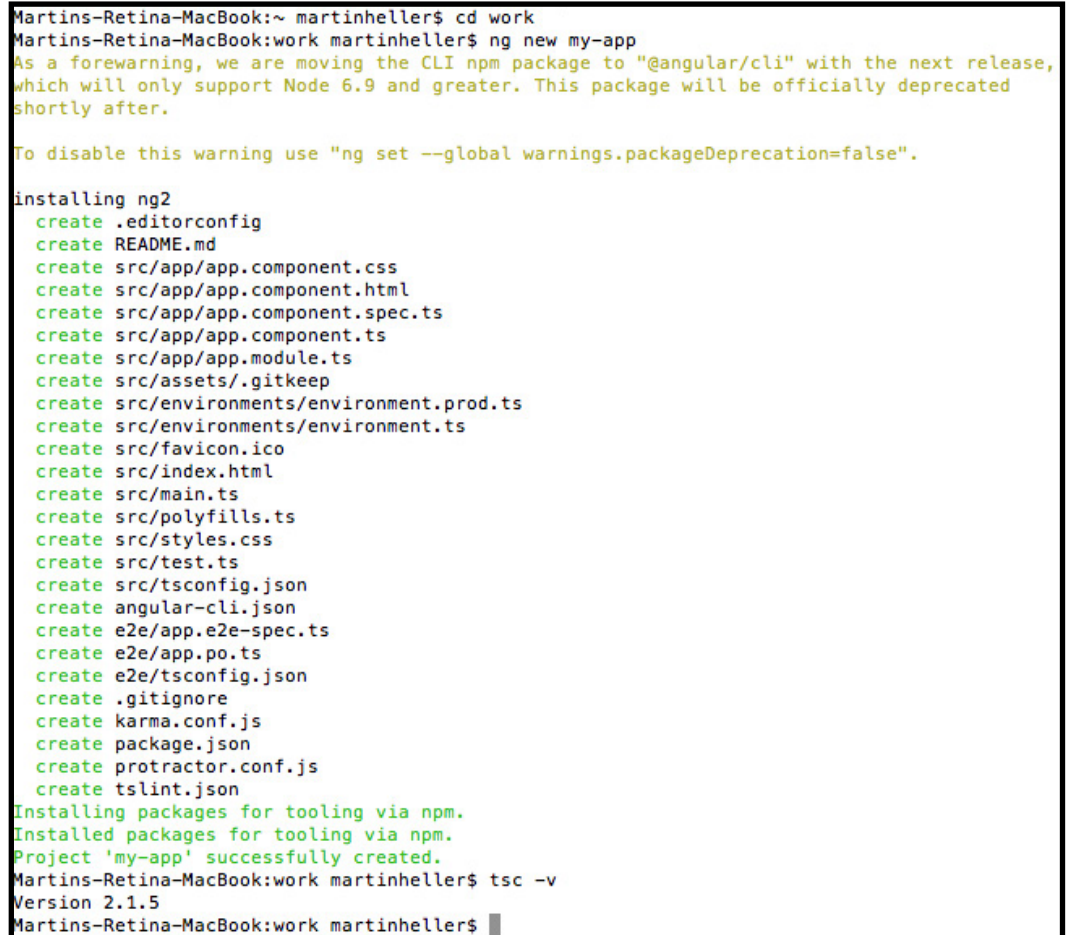
```
$ npm install -g @angular/cli
```

If you watch carefully as the installation proceeds, you'll see a bunch of prerequisites and tools installed before Angular and its CLI. If there are warnings, don't worry about them. If there are errors, you may have to fix them; I've only seen warnings myself. It is safe to reinstall the Angular CLI whenever you wish.

Deep Dive

The next step is to create a new project with the Angular CLI. I put mine inside a directory called Work under my home user folder.

```
$ cd work
$ ng new my-app
```




A terminal window showing the command `ng new my-app` being executed. The output includes a warning about the Angular CLI npm package being deprecated, instructions to disable the warning, and a list of files being created for the new application. The terminal also shows the installation of packages for tooling via npm and the successful creation of the project 'my-app'. Finally, the command `tsc -v` is run to check the TypeScript version, which is 2.1.5.

```
Martins-Retina-MacBook:~ martinhellert$ cd work
Martins-Retina-MacBook:work martinhellert$ ng new my-app
As a forewarning, we are moving the CLI npm package to "@angular/cli" with the next release,
which will only support Node 6.9 and greater. This package will be officially deprecated
shortly after.

To disable this warning use "ng set --global warnings.packageDeprecation=false".

installing ng2
  create .editorconfig
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/assets/.gitkeep
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.json
  create angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Project 'my-app' successfully created.
Martins-Retina-MacBook:work martinhellert$ tsc -v
Version 2.1.5
Martins-Retina-MacBook:work martinhellert$
```



As the instructions note, generating the new Angular app takes a few minutes. This is a good time to go brew a nice cup of tea or coffee.

As the instructions note, generating the new Angular app takes a few minutes. This is a good time to go brew a nice cup of tea or coffee.

You'll see in the screenshot that I double-checked my TypeScript version (**tsc -v**) after the Angular CLI installation. Yes, it was a little paranoid. And yes, it's a good idea for you to do as well. If you have not installed TypeScript already, let's take care of that now:

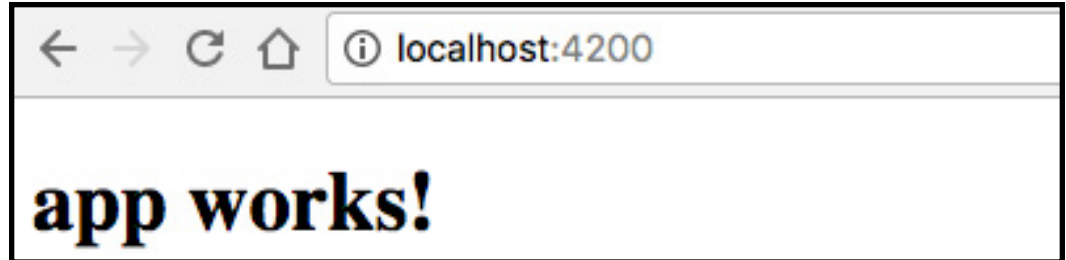
```
$ npm install -g typescript
```

We're almost there. Next, step into the new directory and serve the application.

```
$ cd my-app
$ ng serve
```

Deep Dive

As the server will tell you, it's listening on port 4200. So open a browser tab to `http://localhost:4200/` and you'll see:



The balance of the CLI QuickStart page instructs you to change the title property and its CSS. Feel free to do that with whatever *programming* editor (*not* a word processor!) you happen to have installed, or wait until we install Visual Studio Code later. The browser window will update automatically whenever you save, as the server watches the code and updates on changes.

When you're done with the server, press Control-C in the terminal window to kill the process.

1b. Install the Angular QuickStart seed

Only do this step if you have skipped step 1a. If you do *both* steps, this installation will clobber part of the CLI installation, and you'll have to redo that if you want to use it again.

The instructions for installing the QuickStart seed offer two options to start the process: [downloading the seed](#) and unzipping it, or alternatively cloning the seed, as follows:

```
$ git clone https://github.com/angular/quickstart.git quickstart
```

Whichever option you choose for getting the code, the next steps are the same:

```
$ cd quickstart (or whatever you named the folder)
$ npm install
$ npm start
```

The `npm install` step does essentially the same thing as the `$ npm install -g @angular/cli` step in the CLI version of the installation, except that it does install TypeScript and it does *not* install the Angular CLI, since that isn't on the dependency list in `package.json`. In fact if the Angular CLI is already installed, this script will *uninstall* it.

The `npm start` step runs this script:

```
"start": "concurrently \"npm run build:watch\" \"npm run serve\""
```

To expand that, the `build:watch` and `serve` scripts are:

```
"build:watch": "tsc -p src/ -w" and
"serve": "lite-server -c=bs-config.json"
```

Have I mentioned that `tsc` is the TypeScript compiler? The `-p` option sets the project directory to compile, and the `-w` option says to watch input files.

The `npm start` step (running the two scripts concurrently) will do essentially the same thing as the `ng serve` step in the CLI version of the installation, except that it is likely to choose a different port, plus it will automatically load the page it is serving in your default browser, and the page will look like this:

Deep Dive



When you're finished playing with your Angular QuickStart app, just hit Ctrl+C or close the terminal window to kill the process. You can start it up again by returning to the directory and running **ng serve**.

The next (optional) step in the QuickStart seed instructions is the one that makes me nervous: It tells you to

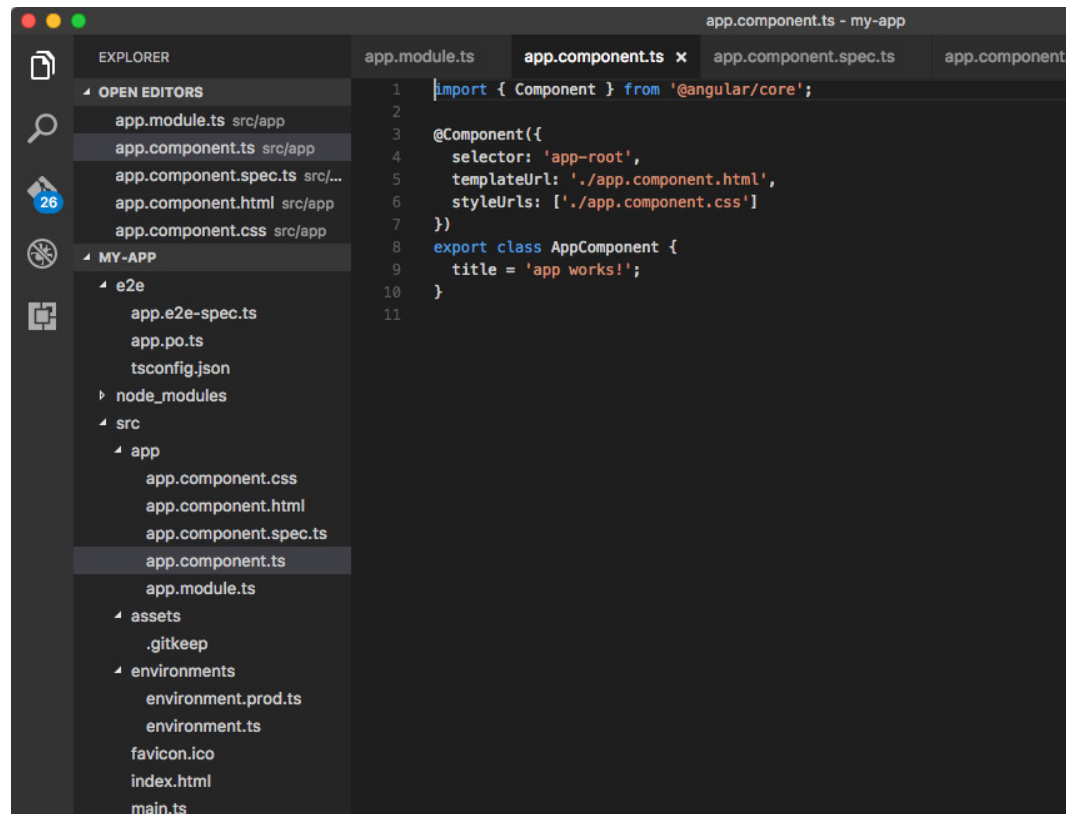
delete the non-essential files using **rm -rf** on MacOS or **del** on Windows. If you decide to do that, at least double-check that you're in the correct directory before firing off the script you copied from the documentation site. Please don't try it after you've started to add files to the project.

No matter whether you followed the CLI or QuickStart seed instructions, your next step will be to explore the source code of an Angular project. To that end, let's install an Angular-aware editor.

2. Install Visual Studio Code

The [Angular resources page](#) recommends three IDEs: IntelliJ IDEA, Visual Studio Code, and WebStorm. I use all three, but for the purposes of this exercise Visual Studio Code is the best choice because it's free and open source. Browse to the [Visual Studio Code home page](#) and download the current stable version for your platform, then install the package.

Once Visual Studio Code is installed, run it and open the directory that holds your base project. On my Mac, the CLI-generated project is at **~/work/my-app** and the seed is at **~/work/quickstart-master**. Your location will vary depending on whether you did the CLI install or the seed install, and any choices you made about their target directories. The source tree should look something like this:



Deep Dive

Visual Studio Code supports TypeScript out of the box, so there's nothing else to install. If you wish to install other languages later, it's easy to do so in the Extensions panel, easily shown by clicking on the bottom icon at the top left. Type the name of the language or tool you want into the search box at the top of the Extensions panel. You can get back to the file explorer by clicking the top icon at the top left.

Note that Visual Studio Code installs a private copy of TypeScript for itself, while Angular installs a global copy of TypeScript. These aren't always the same version, but it shouldn't matter. If Visual Studio Code starts bleating about this, you can click on the button that essentially says "Don't bother me about this again" and it'll change its options. If you want the version check reinstated at any time, go to Code > Preferences > Settings and change the value next to **"typescript.check.tscVersion"** from **false** to **true**.

Explore the Angular source code

Now that we finally have a working Angular "Hello" project open in an editor, we can start exploring the source code. Actually, let's go back to the documentation first and look at the setup guide to see the explanations. There are boxes about halfway down the page, which show the three Angular core source files:

What's in the QuickStart seed?

The **QuickStart seed** contains the same application as the QuickStart playground. But its true purpose is to provide a solid foundation for *local* development. Consequently, there are *many more files* in the project folder on your machine, most of which you can [learn about later](#).

Focus on the following three TypeScript (`.ts`) files in the `/src/` folder.

```

src
├── app
│   ├── app.component.ts
│   └── app.module.ts
└── main.ts
  
```

```

src/app/app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})
export class AppComponent { name = 'Angular'; }
  
```

All guides and cookbooks have at least these core files. Each file has a distinct purpose and evolves independently as the application grows.

Files outside `src/` concern building, deploying, and testing your app. They include configuration files and external dependencies.

Files inside `src/` "belong" to your app. Add new Typescript, HTML, and CSS files inside the `src/` directory, most of them inside `src/app`, unless told to do otherwise.

The following are all in `src/`

File	Purpose
<code>app/app.component.ts</code>	Defines the same <code>AppComponent</code> as the one in the QuickStart playground. It is the root component of what will become a tree of nested components as the application evolves.
<code>app/app.module.ts</code>	Defines <code>AppModule</code> , the root module that tells Angular how to assemble the application. Right now it declares only the <code>AppComponent</code> . Soon there will be more components to declare.

Deep Dive

We've seen **app.component.ts** before, in the online QuickStart. As the page says, "it is the root component of what will become a tree of nested components as the application evolves." The next file over, **app.module.ts**, defines "the root module that tells Angular how to assemble the application. Right now it declares only the **AppComponent**. Soon there will be more components to declare."

The third file, **main.ts**, is deceptively simple.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

platformBrowserDynamic is the Angular JIT compiler, and **bootstrapModule** is the function that kicks off the app. Yes, Angular has *its own just-in-time compiler*, which knows how to turn Angular directives into executable JavaScript. Let's not get into Angular's other compilers at this point.

I told you earlier not to worry about how the decorator function and interpolation binding are implemented. Now you know: Angular supplies its own compilers that turn all the declarations into code.

If you go back to Visual Studio Code and open **src/main.ts**, you'll find essentially the same contents (modulo whitespace) if you used the QuickStart seed. You'll find a longer, more complicated version of the bootstrap file if you generated the app from the Angular CLI, which hints at the different compilers and environments available for Angular:

app.module.ts	app.component.ts	main.ts	x	app.component.spec.ts	app.component.html
1	import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';				
2	import { enableProdMode } from '@angular/core';				
3	import { environment } from './environments/environment';				
4	import { AppModule } from './app/app.module';				
5					
6	if (environment.production) {				
7	enableProdMode();				
8	}				
9					
10	platformBrowserDynamic().bootstrapModule(AppModule);				
11					

What's going on here? We see an environment with a variable called **production**, and a setter function called **enableProdMode**. There's an explanation in the source code found in **environment/environments.ts** in the CLI-generated app:

module.ts	app.component.ts	main.ts	environment.ts	x	app.component.spec.ts	app.component.html	app
1			// The file contents for the current environment will overwrite these during build.				
2			// The build system defaults to the dev environment which uses 'environment.ts', but if you do				
3			// 'ng build --env=prod' then 'environment.prod.ts' will be used instead.				
4			// The list of which env maps to which file can be found in 'angular-cli.json'.				
5							
6			export const environment = {				
7			production: false				
8			};				
9							

Deep Dive



More seriously, the problem with Angular is not the architecture itself, but rather that the terminology is overloaded with JavaScript and TypeScript terminology.

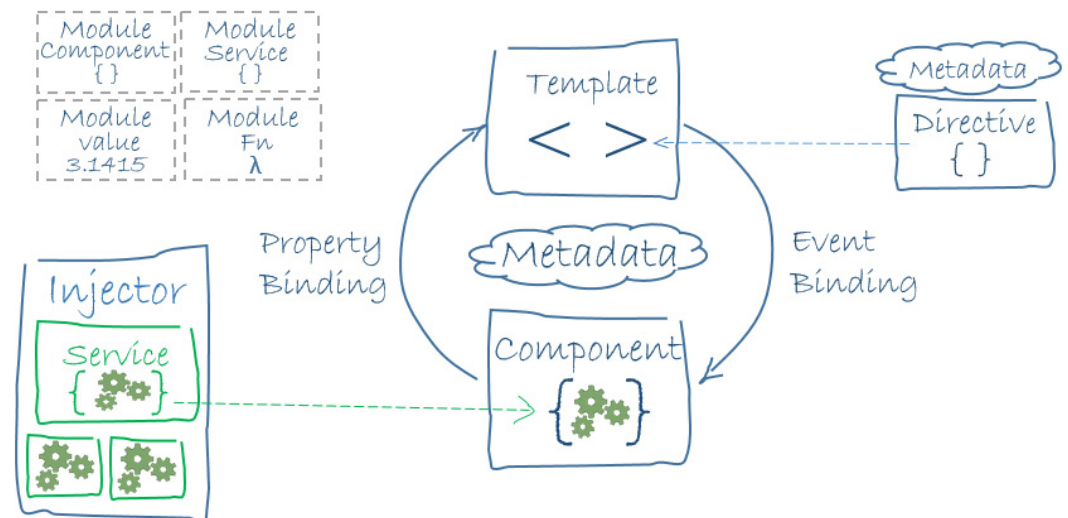
Angular architecture

As I planned this tutorial, I considered sending you to the official [Angular architecture documentation](#) to learn what they had to say about their own architecture; however, I didn't want any damage to your mental health on my conscience. You're free to go look at that, but you may want a stiff drink first.

More seriously, the problem with Angular is not the architecture itself, but rather that the terminology is overloaded with JavaScript and TypeScript terminology. JavaScript (more accurately CommonJS and TypeScript) and Angular both have *modules*, but they are different. JavaScript and Angular both have *libraries*, but again they are different. Both have *imports* and *exports*, and, as you've probably guessed, they are different. Have you reached for the bottle yet? How about the headache pills?

Angular, as I mentioned above, adds *decorators*, *directives*, and *components* to JavaScript and TypeScript. Since none of these are native facilities, Angular has its own compiler to turn them into JavaScript, which you launch, and then have the compiler *bootstrap* your app. Meanwhile, TypeScript also has a compiler, more accurately a transpiler, which generates JavaScript.

OK. Take a deep breath. Let's look at the Angular architecture diagram.



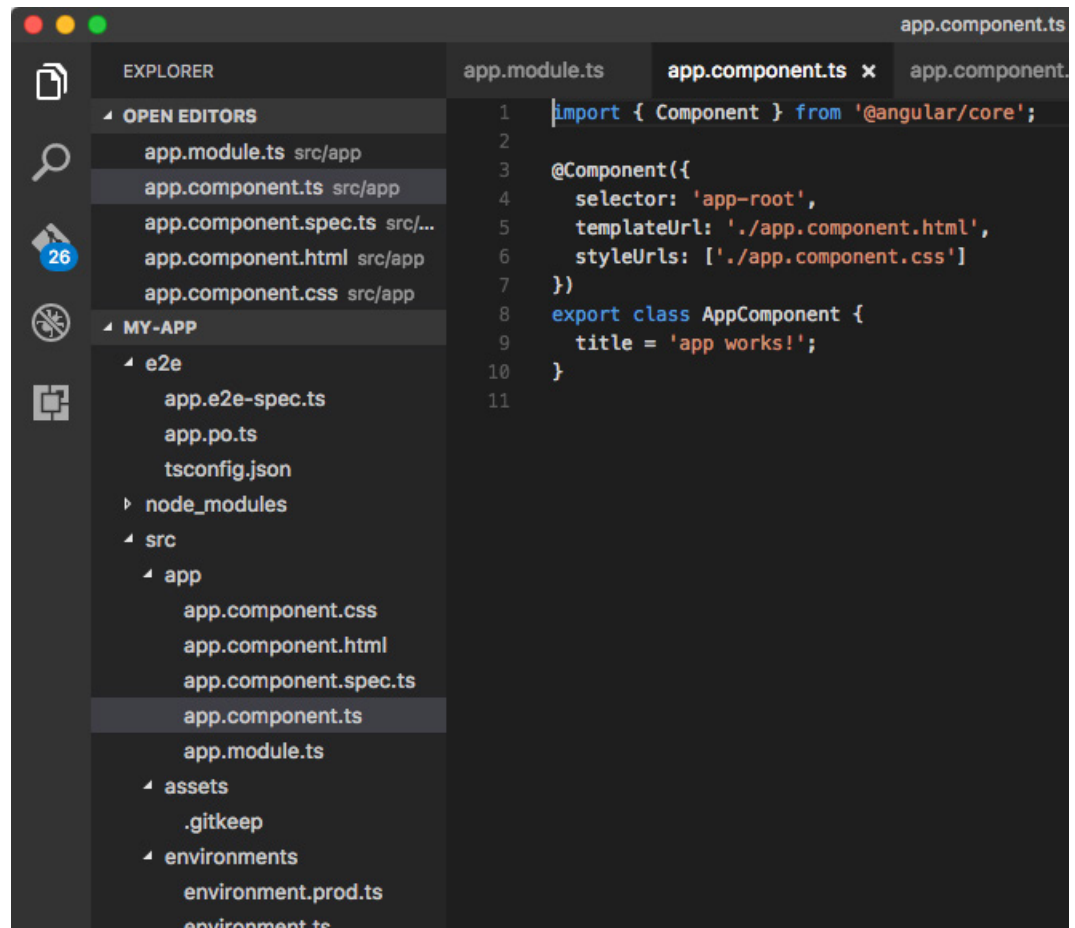
What are we seeing here? Basically, the diagram shows the relationships between some of the major building blocks of Angular: modules, components, templates, metadata, data binding, directives, services, and dependency injection. There are additional optional building blocks in Angular, such as the router module.

For this tutorial, I'm going to concentrate on the four building blocks that you'll need on a daily basis: components, directives, services, and routers. If you feel the need to understand Angular modules right this minute, by all means go [read the documentation](#) and then come back. If not, stay with me.

Components

Let's look again at `app.components.ts`, which is the *root component* of an Angular app. For a generated seed app, it's the *only* component.

Deep Dive



There are several ways to think about Angular components. One is that they are TypeScript classes, as seen at line 8 above. Note that classes are exported in TypeScript. Since by convention Angular components are in separate files, TypeScript needs to know that you intend them to be visible to other components.

No other component actually needs to import the base class, but we export it anyway as a matter of consistency.

```
export class AppComponent {
  title = 'app works!';
}
```

Another way to think about components is that they each control a patch of screen called a view, which starts to make sense when an app has many components for specialized views, such as a top bar, a side navigation bar, a side toolbar, and a tabbed detail view. All of those view components together could describe Visual Studio Code if it were an Angular app. (It's not.)

A third way to look at Angular view components is as the container for all the metadata that ties the view together, expressed as a **@Component directive**. The root component shown above defines three metadata properties:

```
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
```

Deep Dive

The *selector* defines the mapping from an Angular view to the DOM (Document Object Model), and says to display the view inside the (custom) `<app-root>` tag in `index.html`:

```
<body>
  <app-root>Loading...</app-root>
</body>
```

In Angular you don't mess with HTML selectors from your code, as you would in jQuery. Instead you set properties and values in the component class. The data binding automatically reflects them to the DOM.

The `templateUrl` property defines the name and location of the Angular template file for the view. The template combines HTML markup with Angular template syntax, as shown by the HTML tags and the handlebar notation below:

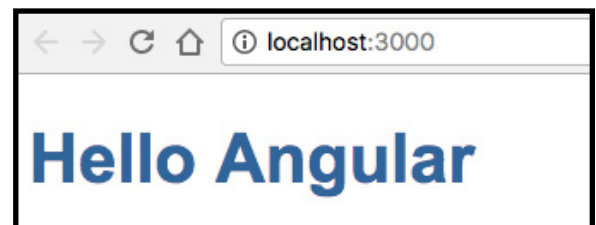
```
<h1>
  {{title}}
</h1>
```

As you have probably guessed, this template says to display the `AppComponent` class `title` property as a level-1 head. We saw "app works!" in a basic H1 style when we ran the app earlier. This could also have been done with an embedded template property and string; separating the `template` out into its own file using the `templateURL` property helps with maintainability and is more convenient for templates of non-trivial size. (I was going to say [rodents — I mean templates — of unusual size](#), but in fact Angular templates are *usually* bigger than you'd want to stuff onto one line.)

The `styleUrls` property holds an array of CSS file names and locations to define the styles that apply to the view. It needs to be an array because CSS styles are hierarchical. As it happens, the default `app.component.css` file in a generated seed app is empty, which is why we saw a basic H1 style and not something better styled. Later in the Angular CLI quick start tutorial, you change the title text and set the CSS to:

```
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
```

That gives you the same look as the seed project, which also has title text of "Hello Angular."



Directives

The Angular `@Component` we've been discussing is a special kind of *directive*. Angular renders templates according to the instructions given by directives, and dynamically transforms the DOM.

There are two other kinds of directives besides components: *structural* and *attribute* directives. Structural directives alter layout by adding, removing, and replacing elements in the DOM. Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

Let's look at some examples from the Angular documentation. The first example, from the Hero List template, includes two built-in structural directives.

Deep Dive

```
<li *ngFor="let hero of heroes"></li>
...
<hero-detail *ngIf="selectedHero"></hero-detail>
```

The top example, ***ngFor** written inside list tags, generates one list item for every **hero** in the **heroes** list. This is a very concise, expressive way to create a list based on data; Angular's JIT compiler turns it into efficient JavaScript and HTML.

The second example, ***ngIf** written inside a custom **hero-detail** tag, only displays a **hero-detail** record if one has been selected from the heroes list and the selected detail record exists. Together with the ***ngFor** directive, this efficiently implements a master-detail application. For the full context, consider the entire Hero List template:

```
<h2>Hero List</h2>
<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>
<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>
```

The **ngModel** directive, which implements two-way data binding, is an example of an **attribute** directive. **ngModel** modifies the behavior of an existing element (typically an **<input>** tag) by setting its display value property and responding to change events. Here's how the **hero-detail** component uses **ngModel** to handle changes:

```
<input [(ngModel)]="hero.name">
```

The magic here is that Angular not only sets the initial value of the **input** field, it takes care of updating the model in response to change events, without making you write event handler code.

Much of the routine work in developing Angular applications involves writing custom components. You can also write custom structural and attribute directives when and if you need them.

Services

When I discussed the Hero List above, I glossed over how the component gets its list of heroes. In fact, it uses a service provider that implements that functionality. Sure, the component could technically implement everything itself, but the preferred style for Angular applications is to keep components focused on the view and the model, with all the back-end work consumed as services.

Service is a broad category in Angular that includes any value, function, or feature that your application needs. The **HeroService** is an exported class that maintains an internal list of heroes, which it gets from a **BackendService** that implements a **getAll** method:

```
export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }
```

Deep Dive

```
getHeroes() {
  this.backend.getAll(Hero).then( (heroes: Hero[]) => {
    this.logger.log('Fetched ${heroes.length} heroes. ');
    this.heroes.push(...heroes); // fill cache
  });
  return this.heroes;
}
}
```

How does the **HeroService** know how to use the **BackendService** and the **Logger**? In the root module, preferentially, these and any other services are listed as providers:

```
providers: [
  BackendService,
  HeroService,
  Logger
],
```

You can also register a service at the component level in the **providers** property of the **@Component** metadata:

```
@Component({
  moduleId: module.id,
  selector: 'hero-list',
  templateUrl: './hero-list.component.html',
  providers: [ HeroService ]
})
```

The preference for registering services in the root module is driven by wanting all components to use the same instance of the services. You might want to register a service at the component level to keep all the pieces of the component together, for modularity, or to give each instance of the component its own instance of a service.

A component lists the services it needs in its *constructor*:

```
constructor(private service: HeroService) { }
```

When Angular creates a component, it first asks an *injector* for the services that the component requires. The injector keeps all its service instances in a container. At component instantiation time, Angular calls the injector, gets the service instances, and then calls the component's constructor with the instances as arguments. That, in a nutshell, is *dependency injection*, which is one way to accomplish *inversion of control*.

Routers

I mentioned earlier that Angular applications usually contain multiple view components. Once you have more than one view, you need a way to navigate from one view to another. For that Angular has *routers*, implemented in an external, optional Angular NgModule called **RouterModule**. The router is a combination of multiple provided services (**RouterModule**), multiple directives (**RouterOutlet**, **RouterLink**, **RouterLinkActive**), and a configuration (**Routes**).

Why not just switch from view to view using HTML links? The basic reason is that links cause a

Deep Dive

new page to load, and Angular by preference creates *single-page applications* (SPAs), which appear smoother to the user.

Routing in a single-page app, whether it's an application based on Angular or another technology such as Ruby on Rails, can get complicated fairly quickly. We want to preserve the idea of a mapping between a URL and an application state without incurring the time overhead of a complete page reload after every HTTP request.

In Angular, that requires a number of conditions to be met. First, the base HREF must be set explicitly:

```
<head>
  <base href="/">
```

Next, we need to map routes between paths and components:

```
import { RouterModule } from '@angular/router';
RouterModule.forRoot([
  {
    path: 'heroes',
    component: HeroesComponent
  }
])
```

There's only one path here now, but there will be more.

Third, we need to add that to the AppModule **imports** array. Fourth, we need to set router links and outlets, to emulate what happens with an HTML link:

```
template: `
  <h1>{{title}}</h1>
  <a routerLink="/heroes">Heroes</a>
  <router-outlet></router-outlet>
`
```

Finally, we want to redirect the null path to our link:

```
{
  path: '',
  redirectTo: '/heroes',
  pathMatch: 'full'
},
```

If you follow the full [Tour of Heroes tutorial](#), you'll see that it adds a dashboard view and sets the **redirectTo** to point to **/dashboard**, which causes the routing to actually make sense. In fact, at this point, I think that following the full tutorial is exactly the right thing to do. You are ready to take it on.

Have fun! ■

Martin Heller is a contributing editor and reviewer for InfoWorld. Formerly a web and Windows programming consultant, he developed databases, software, and websites from his office in Andover, Massachusetts, from 1986 to 2010. More recently, he has served as VP of technology and education at Alpha Software and chairman and CEO at Tubifi.