****

**Angular 7**

**Submitted By:**

**Shaikh Sameer**

### What’s new in Angular 7

* Angular 7 now supports Typescript 3.1, RxJS 6.3 and Node 10.
* While creating a new Angular application, the Angular CLI prompts users to add features like routing or SCSS support. The ng new capability asks for routing and what type of styles to use, while ng add @angular/material asks for a theme, gestures, and animations.
* Angular 7 will automatically remove reflect-metadata polyfill from your polyfills.ts file, as it is only meant for development, not for production.
* Angular 7 now uses Angular CLI Bundle Budget feature.

Budgets is a feature in the Angular CLI which allows you to set budget thresholds in your configuration to ensure parts of your application stay within boundaries which you set.

New applications will warn when the initial bundle is more than 2MB (default value) and will error at 5MB. This budget feature is configurable and easy to change in your angular.json.

|  |  |
| --- | --- |
|  |  |

* Angular Material & the CDK also received some major updates. The two newly added features of CDK are Virtual Scrolling and Drag and Drop.  
  **1. Virtual Scrolling**

Virtual Scrolling loads and unloads elements from the DOM based on the visible parts of a list, making it possible to build very fast experiences for users with very large scrollable lists.

**2. Drag and drop**

Drag and drop support is now in the CDK and includes automatic rendering as the user moves items and helper methods for reordering lists (moveItemInArray) and transferring items between lists (transferArrayItem).

* The CLI documentation has been integrated into the main Angular Docs docs.
* Angular Elements now supports content projection using web standards for custom elements.

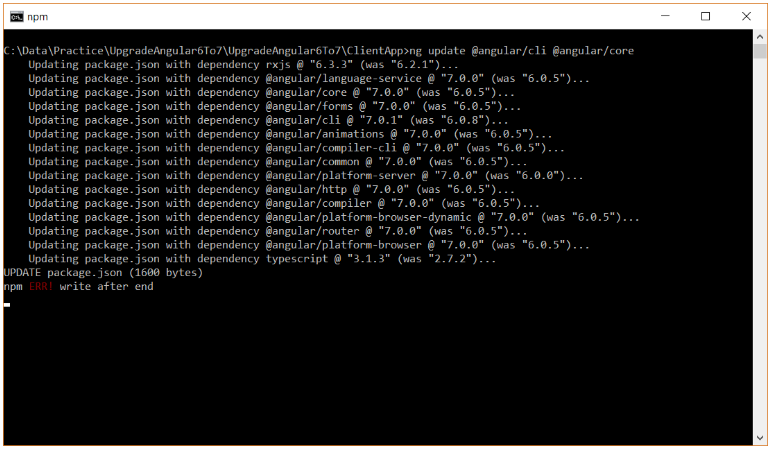
|  |  |
| --- | --- |
|  |  |

### Upgrade Angular 6 app to Angular 7

The upgrade to angular 7 from 6 is easy and it can be completed in one command. Thanks to the work done around tooling in Angular 6. To upgrade to Angular 7, navigate to your Angular 6 app directory (where package.json is present) and run the following command.

|  |  |
| --- | --- |
| 1 | ng update @angular/cli @angular/core |

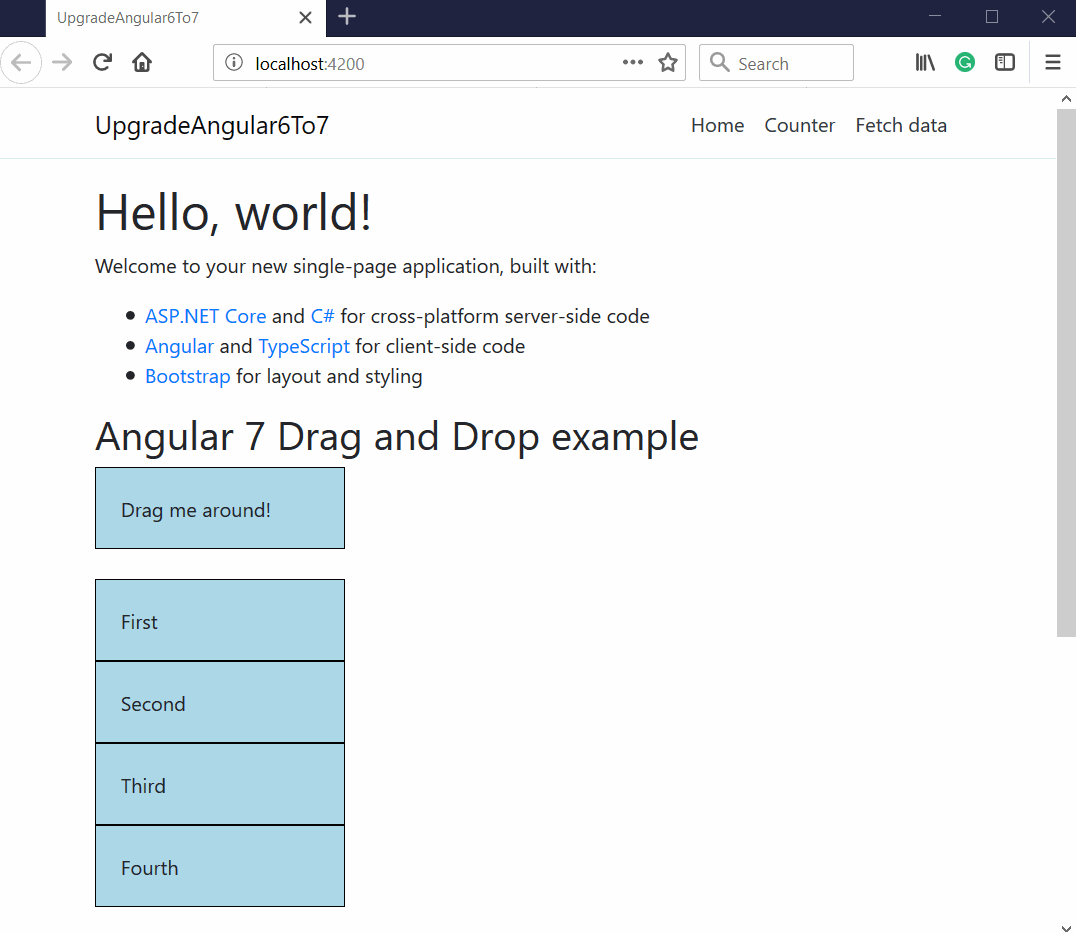
You should see the following. All your packages are now updated to Angular 7. You can verify it via the package.json file.



Build and run the application to see if it is running properly, after the upgrade. If you face any issue, visit [update.angular.io](http://update.angular.io/) for detailed information and guidance on updating your application.

### Implement Angular 7 Drag and Drop Feature

As mentioned earlier, Angular 7 supports **drag and drop feature** which allows you to create drag-and-drop interfaces, with support for free dragging, sorting within a list, transferring items between lists, animations, touch devices, custom drag handles, previews. Take a look at below GIF to get an idea about Drag and Drop feature.



* First, we need to install the angular cdk package and import the DragDropModule module. Install the module via npm:

|  |  |
| --- | --- |
| 1 | npm install @angular/cdk@latest |

* Once the module is installed, run ng serve command to check the application is running properly or not. You may get the following error while running the ng serve command
* **“You seem to not be depending on “@angular/core” and/or “rxjs”. This is an error.”**.
* To fix this error, first run npm link and then run ng serve command.

|  |  |
| --- | --- |
| 1  2 | npm link  ng serve |

* Next, import the DragDropModule module. To do that , open app.module.ts and make the highlighted code changes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpClientModule } from '@angular/common/http';  import { RouterModule } from '@angular/router';  import { DragDropModule } from '@angular/cdk/drag-drop';    import { AppComponent } from './app.component';  import { NavMenuComponent } from './nav-menu/nav-menu.component';  import { HomeComponent } from './home/home.component';  import { CounterComponent } from './counter/counter.component';  import { FetchDataComponent } from './fetch-data/fetch-data.component';    @NgModule({    declarations: [      AppComponent,      NavMenuComponent,      HomeComponent,      CounterComponent,      FetchDataComponent    ],    imports: [      BrowserModule.withServerTransition({ appId: 'ng-cli-universal' }),      HttpClientModule,      FormsModule,      DragDropModule,      RouterModule.forRoot([        { path: '', component: HomeComponent, pathMatch: 'full' },        { path: 'counter', component: CounterComponent },        { path: 'fetch-data', component: FetchDataComponent },      ])    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

* Add the following HTML in home.component.html file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <h2>Angular 7 Drag and Drop example</h2>  <div class="box" cdkDrag>Drag me around!</div>  <br />  <div cdkDropList (cdkDropListDropped)="onDrop($event)">    <div class="box" \*ngFor="let item of items" cdkDrag>{{item}}</div>  </div> |

* Few points worth mentioning here:
  + You can create draggable component by using the cdkDrag directive.
  + You can add cdkDropList elements to constrain where elements may be dropped.
  + Adding cdkDropList around a set of cdkDrag elements groups the draggables into a reorderable collection. Items will automatically rearrange as an element moves. **Note** that this will not update your data model; you can listen to the **cdkDropListDropped** event to update the data model once the user finishes dragging.
* Next, add these following CSS class in the style.css file. These classes styles the elements and provides animation while dragging and dropping.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | .box {    border: 1px solid black;    padding: 20px;    width: 200px;    background-color:lightblue;  }    .cdk-drop-dragging .cdk-drag {    transition: transform 500ms cubic-bezier(0, 0, 0.2, 1);  }    .cdk-drag-animating {    transition: transform 550ms cubic-bezier(0, 0, 0.2, 1);  }    .cdk-drag-placeholder {    background-color: Highlight;  } |

* Finally, open home.component.ts and place the highlighted code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import { Component } from '@angular/core';  import { CdkDragDrop, moveItemInArray } from '@angular/cdk/drag-drop';    @Component({    selector: 'app-home',    templateUrl: './home.component.html',  })  export class HomeComponent {    items = ['First', 'Second', 'Third', 'Fourth'];      onDrop(event: CdkDragDrop<string[]>) {      moveItemInArray(this.items, event.previousIndex, event.currentIndex);    }  } |

* As you can see in the code snippet above, the drag and drop CDK comes with a function moveItemInArray. This function rearranges the items array according to the indexes. The onDropevent is called, when the item is dropped.

That’s it. Run the app and you should be able to drag and drop the object as shown in the earlier image earlier.

How to update to v7

Visit [update.angular.io](https://update.angular.io/) for detailed information and guidance on updating your application, but thanks to the work we did in v6, updating to v7 should be one command for most developers:

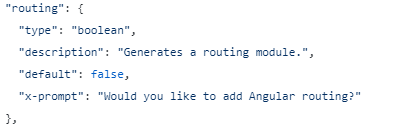
*ng update @angular/cli @angular/core*

Early adopters of v7 have reported that this update is faster than ever, and many apps take less than 10 minutes to update.

### CLI Prompts

The CLI will now prompt users when running common commands like ng new or ng add @angular/material to help you discover built-in features like routing or SCSS support.

CLI Prompts have been added to [Schematics](https://blog.angular.io/schematics-an-introduction-dc1dfbc2a2b2), so any package publishing Schematics can take advantage of them by adding an x-prompt key to a Schematics collection.



### Application Performance

Continuing our focus on performance, we analyzed common mistakes across the ecosystem. We discovered that many developers were including thereflect-metadata polyfill in production, which is only needed in development.

To fix this, part of the update to v7 will automatically remove this from your polyfills.ts file, and then include it as a build step when building your application in JIT mode, removing this polyfill from production builds by default.

With v7, we are also defaulting new projects to take advantage of Bundle Budgets in our CLI. New applications will warn when the initial bundle is more than **2MB** and will error at **5MB**. These budgets are easy to change in your angular.json.

### Understanding Schematics

In a schematic, you don’t actually perform any direct actions on the filesystem. Rather, you describe what transformation you would like to apply to a Tree. This allows us to support features like dry runs (or patch runs) without adding special support from the schematics themselves. It also makes schematics hermetic which ensures reusability and safety.

The Tree is a data structure that contains a base (a set of files that already exists) and a staging area (a list of changes to be applied to the base). When making modifications, you don’t actually change the base, but add those modifications to the staging area. This is really powerful but can be tricky and will be further explored in a separate medium post.

The Tree that a schematic will receive can be anything. The Angular CLI will use a Tree representing the project on the drive to the first schematic it calls, but composed schematics could receive any Trees. The good news is that it doesn’t matter; the Tree represents your starting point.

### Creating your first Schematics

First, make sure you have Node 6.9 or above installed. Next, install the Schematics command line tool globally:

npm install -g @angular-devkit/schematics-cli

This will install a schematics executable, which you can use to create a blank Schematics project:

schematics blank --name=my-component

Et voilà. The blank schematics either create a new project, or add a blank schematic to an existing project (it can be used for both). You can then cdinto your new project, install your npm dependencies, and open your new collection using your favorite editor of choice:

cd my-component  
npm install

### Collections

Schematics Collections are sets of named schematics, that are published and installed by users. For example, the Angular team publishes and maintains the official @schematics/angular collection, which contains schematics like component, module and application.

In our case, our collection will only include the my-component schematic. You can look at the src/collection.json file, which contains the description of our collection:

{  
 "$schema": "../node\_modules/@angular-devkit/schematics/collection-schema.json",  
 "schematics": {  
 "my-component": {  
 "description": "A blank schematic.",  
 "factory": "./my-component/index#myComponent"  
 }  
 }  
}

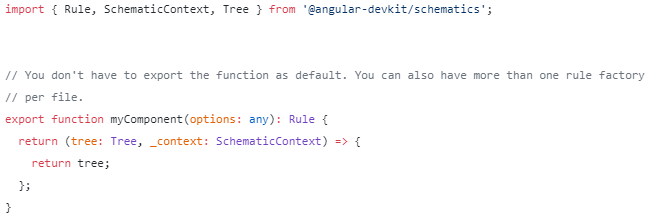
The $schema key points to the JSON Schema defining this format. It is used by IDEs to do auto completion, tools for validation, and is entirely optional.

The important key is "schematics", which describes the schematics included in this collection. In our example, we describe one schematic: my-component, it has a simple description and a factory field. The factory field uses a string reference to point to a JavaScript function; in our case the exported function myComponent in the file my-component/index.js. It represents the RuleFactory.

### Rules, Trees and Files

A Rule is a function that takes a Tree and returns another Tree. Rules are the core of Schematics; they are the ones making changes to your projects, calling external tools, and implementing logic. RuleFactory, as the name implies, are functions that create a Rule.

Here’s the blank RuleFactory created so far:



This factory takes an options argument and returns a Rule that takes a Tree and returns it unchanged.

The options argument is an object that can be seen as the input of the factory. From the CLI, it is the command line arguments the user passed. From another schematic, it’s the options that were passed in by that schematic. A GUI tool could construct an options object from user or project inputs, for example.

In any case, this is always an object and can be typed as any. It can also be validated with a JSON Schema to make sure that the inputs have appropriate default and types. JSON Schemas will be looked at more closely in a later post.

In the meantime, let’s do something more interesting with our rule:



With this new line, we’re creating a file in the root of the schematic’s Tree, named either after the name option (or 'hello' by default), containing the string world. This might seem trivial for now, but there’s a lot going on here under the hood.

A Tree contains the files that your schematics should be applied on. It has a list of files, and contains metadata associated with the changes you want to apply. In our case, the only change being made is to create a new file. Trees are more complex than just being a filesystem equivalent, and will be explored more deeply in a later post, but for the moment you can see them as a collection of files and changes.

By default, the Angular CLI will pass the root of your Angular project as the Tree, but any schematic can pass in a different Tree to other schematics. You can create empty trees, scope a Tree to a directory of a parent Tree, merge two trees, or branch them (making a copy of it).

There are four methods that directly create a change in a Tree; create, delete, rename, and overwrite.

### Running Your New Schematics

To run our example, you first need to build it, then use the schematicscommand line tool with the path to our schematic project’s directory as the collection. From the root of our project:

npm run build  
# ... wait for build to finish

schematics .:my-component --name=test  
# ... see that a file is created in the root.

Before looking further into what happens here, a word of warning; don’t worry, this time you did not actually create a file on your filesystem. This is because the schematics tool is in debug mode when using a path as the collection it should use. When in debugging (which can also be used with --debug=true), the default is also to run in dry run mode, which prevents the tool from actually creating files.

This can be changed using the argument --dry-run=false. But beware, this means that the changes will really happen on the filesystem. If you delete or overwrite a file, you might lose content you don’t want to. **We suggest to be in a separate temporary directory when debugging schematics, and to disable dry runs only when necessary.**

You can also start npm run build -- -w in a separate terminal so it automatically rebuild your schematic project when a file changes.

### Debugging

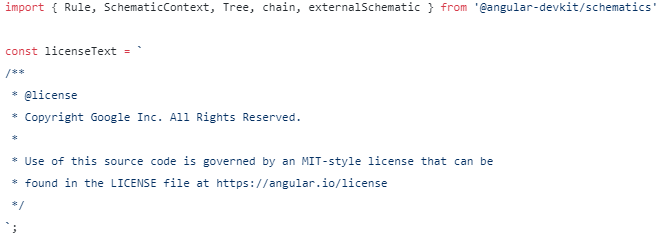
In order to debug your schematics, you need to run with node in debugging mode:

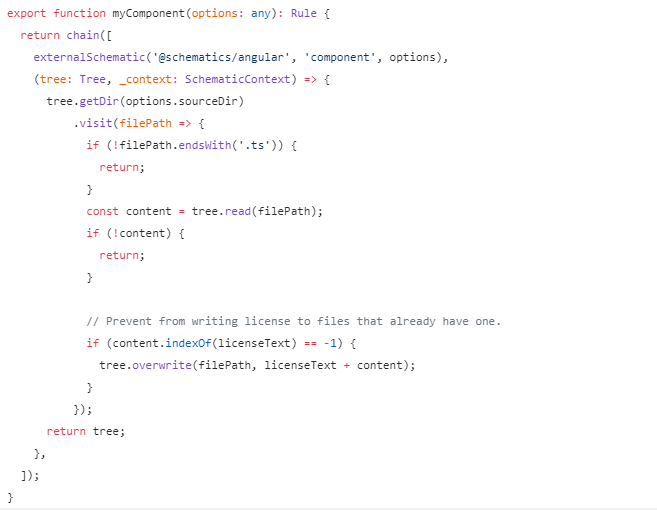
node --inspect-brk $(which schematics) .:myComponent --name=test

Another advantage of running in debug mode is that the schematicscommand line tool puts a break point directly before running your own schematic.

### Calling Another Schematic

One of the great advantage of Schematics is in how easy they are to compose together. In our example, we’ll call the component schematic from the Angular collection to add a component to your application, then add a header to every TypeScript files added by the Schematic.





Don’t forget to add @schematics/angular to your dependencies in your package.json!

A few things to note here. First, we are calling and returning chain() directly. chain() is a RuleFactory provided by the Schematics library that chain multiple rules together, waiting in between for the Rule to finish. There are other rule factories like this provided by the Schematic library, and we’ll go over them at a later moment.

Second, we’re using another RuleFactory called externalSchematic (it also has a sister factory called schematic). Schematics being rules, you might be tempted to simply import a schematic’s rule factory and create the rule yourself, then call it directly (or pass it to chain directly). **Do not call other Schematics as Rules directly.** There is more logic to the externalSchematic(and schematic) rule factory than importing the schematic and running it. For example, validating the Schema and filling default values.

Finally, there are no good way for now to list files that were created or overwritten in a tree. Because Schematics was built to be hermetic, the Treethat you receive does not have local changes. Because of this, we have to go through all files.

### Using Angular CLI

The best usage of Schematics for your users is currently through the Angular CLI. This means you should probably give it a try before publishing this to NPM. Here we will try to use our new myComponent schematic through the Angular CLI.

First, create an empty project with the CLI:

ng new my-project

Then in your new project, link the Schematics we just built:npm link $PATH\_TO\_SCHEMATIC\_PROJECT

Replace $PATH\_TO\_SCHEMATIC\_PROJECT with the path to your project’s root. Note that users will install instead of linking, this is just to iterate faster locally while developing.

Once your schematic project is linked, you can use ng generate to call your schematics:

ng generate my-component:my-component someName

By default, if the schematic takes a name argument, the second argument of the generate command will be set to that name.

Voilà! This should be enough to get your users started. Please note that there’s also a default collection in the CLI configuration that you can set

## Angular 7 Routing

Now that we have a navigation, let's make our little app actually navigation between our components as needed.

Open up **/src/app/app-routing.module.ts** and specify the following contents:

// Other imports removed for brevity

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

import { ContactComponent } from './contact/contact.component';

const routes: Routes = [

{ path: '', component: HomeComponent },

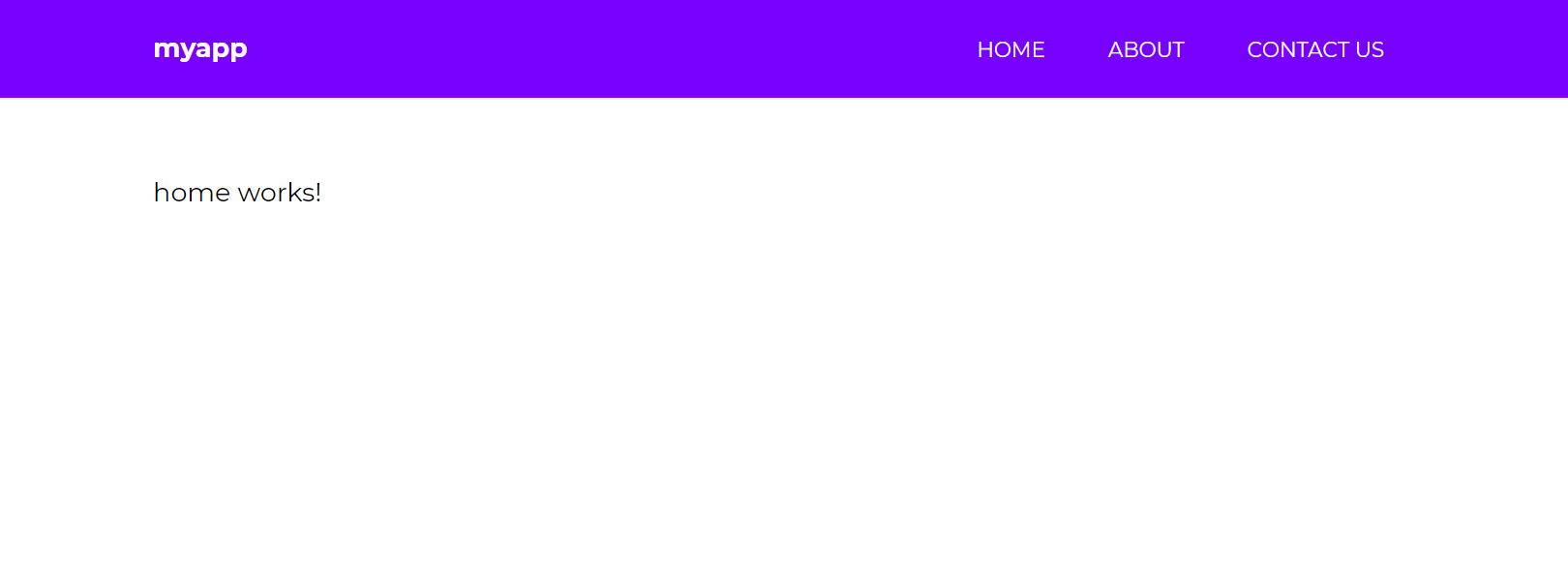
{ path: 'about', component: AboutComponent },

{ path: 'contact', component: ContactComponent },

As we can see here, we're defining importing our components and defining an object for each route inside of the **routes** constant. These route objects also accept other properties, which allow you to define URL parameters, but because our app is simple, we won't be doing any of that.

Save this file and try clicking on the links above. You will see that each of the respective component's HTML templating shows up in the **<router-outlet></router-outlet>** defined in app.component.html.

This is what the result should look like in the browser at this point:



You now know enough about Angular 7 to create a very simple website with routing!  But let's learn more than that.

## Angular 7 Event Binding

In the next several sections, we're going to use our **/src/app/home** component as a playground of sorts to learn features specific to Angular 7.

One of the most used forms of event binding is the click event. You often need to make your app respond when a user clicks something, so let's do that!

Visit the **/src/app/home/home.component.html** template file and specify the following:

<h1>Home</h1>

<button (click)="firstClick()">Click me</button>

You define an event binding by wrapping the event between (), and calling a method. You define the method in the **home.component.ts** file as such:

export class HomeComponent implements OnInit {

constructor() { }

ngOnInit() {

}

firstClick() {

console.log('clicked');

}

}

Save it, get out the browser console (CTRL+SHIFT+i) and click on the button. The output should show clicked. Great!

## Angular 7 Class & Style Binding

Sometimes you may need to change the appearance of your UI from your component logic. There are two ways to do this, through class and style binding.

There are a lot of different methods you can use to control class binding, so we won't cover them all. But I will cover some of the most common use cases.

Let's say that you want to control whether or not a CSS class is applied to a given element. Update the **h1** element in home.component.html to the following:

<h1 [class.gray]="h1Style">Home</h1>

Here, we're saying that the CSS class of .gray should only be attached to the h1 element if the property h1Style results to true. Let's define that in the home.component.ts file:

h1Style: boolean = false;

constructor() { }

ngOnInit() {

}

firstClick() {

this.h1Style = true;

}

Let's also define the .gray class in this component's css file:

.gray {

color: gray;

}

Save it, and you can now click on the click me button to change the color of the Home title.

What if you wanted to control multiple classes on a given element? You can use ngClass. Modify the home component's template file to the following:

<h1 [ngClass]="{

'gray': h1Style,

'large': !h1Style

}">Home</h1>

Then, add the large ruleset to the .scss file:

.large {

font-size: 4em;

}

Now give it a shot in the browser. Home will appear large, but shrink down to the regular size when you click the button. Great!

You can also control appearance by changing the styles directly from within the template. Modify the template as such:

<h1 [style.color]="h1Style ? 'gray': 'black'">Home</h1>

Refresh and give this a shot by clicking the button.Like ngClass() there's also an ngStyle() that works the same way:

<h1 [ngStyle]="{

'color': h1Style ? 'gray' : 'black',

'font-size': !h1Style ? '1em' : '4em'

}">Home</h1>

## Angular 7 Services

Services in Angular 7 allow you to define code that's accessible and reusable throughout multiple components. A common use case for services is when you need to communicate with a backend of some sort to send and receive data.

> ng generate service data

Open up the new service file **/src/app/data.service.ts** and let's create the following method:

// Other code removed for brevity

export class DataService {

constructor() { }

firstClick() {

return console.log('clicked');

}

}

To use this in a component, visit **/src/app/home/home.component.ts** and update the code to the following:

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

import { DataService } from '../data.service';

@Component({

selector: 'app-home',

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

styleUrls: ['./home.component.scss']

})

export class HomeComponent implements OnInit {

constructor(private data: DataService) { }

ngOnInit() {

}

firstClick() {

this.data.firstClick();

}

}

There are 3 things happening here:

* We're first importing the DataService at the top.
* We're creating an instance of it through dependency injection within the constructor() function.
* Then we call the method with this.data.firstClick() when the user clicks on the button.

If you try this, you will see that it works as clicked will be printed to the console. Awesome! This means that you now know how to create methods that are accessible from any component in your Angular 7 app.

## Angular 7 HTTP Client

Angular comes with its own HTTP library that we will use to communicate with a fake API to grab some data and display it on our home template. This will take place within the data.service file that we generated with the CLI.

In order to gain access to the HTTP client library, we have to visit the **/src/app/app.module.ts** file and make a couple changes. Up until this point, we haven't touched this file, but the CLI has been modifying it based on the generate commands we've issued to it.

Add the following to the imports section at the top:

// Other imports

import { HttpClientModule } from '@angular/common/http';

Next, add it to the imports array:

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule, // <-- Right here

],

Now we can use it in our **/src/app/data.service.ts** file:

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

import { HttpClient } from '@angular/common/http'; // Import it up here

@Injectable({

providedIn: 'root'

})

export class DataService {

constructor(private http: HttpClient) { }

getUsers() {

return this.http.get('[**https://reqres.in/api/users**](https://reqres.in/api/users)')

}

}

[**reqres.in**](http://reqres.in/) is a free public API that we can use to grab data.

Open up our home.component.ts file and modify the following:

export class HomeComponent implements OnInit {

users: Object;

constructor(private data: DataService) { }

ngOnInit() {

this.data.getUsers().subscribe(data => {

this.users = data

console.log(this.users);

}

);

}

}

The first thing you might notice is that we're placing the code inside of the **ngOnInit()** function, which is a lifecycle hook for Angular. Any code placed in here will run when the component is loaded.

We're defining a **users** property, and then we're calling the .getUsers() method and subscribing to it. Once the data is received, we're binding it to our users object and also console.logging it.

Give it a try in the browser and you will see the console shows an object that's returned. Let's display it on our home template!

Open up **home.component.html** and specify the following:

<h1>Users</h1>

<ul \*ngIf="users">

<li \*ngFor="let user of users.data">

<img [src]="user.avatar">

<p>{{ user.first\_name }} {{ user.last\_name }}</p>

</li>

</ul>

Great! Let's specify some CSS to make this look better in **home.component.scss**:

ul {

list-style-type: none;

margin: 0;padding: 0;

li {

background: rgb(238, 238, 238);

padding: 2em;

border-radius: 4px;

margin-bottom: 7px;

display: grid;

grid-template-columns: 60px auto;

p {

font-weight: bold;

margin-left: 20px;

}

img {

border-radius: 50%;

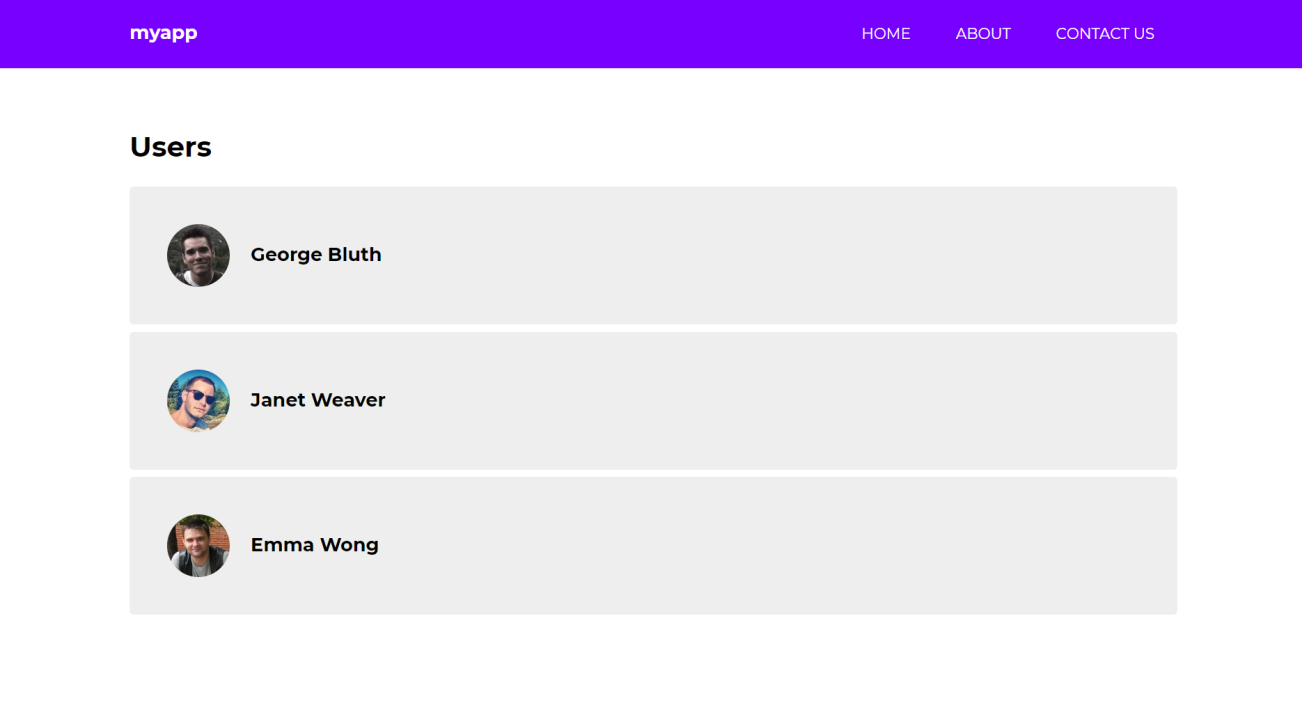
width: 100%;

}

}

}

View the result in the browser:



Awesome!

## Angular 7 Forms

If you recall, we generated a component called **contact**. Let's create a contact form so that you can learn how to use forms in Angular 7.

Angular 7 provides you with two different approaches to dealing with forms: template driven and reactive forms. I'm not going to go into the differences between these two approaches, but reactive forms generally provide you with more control andform validation can be unit tested as opposed to template driven forms.

To get started, we have to visit the **app.module.ts** file and import the Reactive Forms Module:

// other imports

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

// other code

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule,

ReactiveFormsModule // <- Add here

],

 Next, visit the **contact.component.ts** file and specify the following

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

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

selector: 'app-contact',

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

styleUrls: ['./contact.component.scss']

})

export class ContactComponent implements OnInit {

messageForm: FormGroup;

submitted = false;

success = false;

constructor(private formBuilder: FormBuilder) { }

ngOnInit() {

this.messageForm = this.formBuilder.group({

name: ['', Validators.required],

message: ['', Validators.required]

});

}

onSubmit() {

this.submitted = true;

if (this.messageForm.invalid) {

return;

}

this.success = true;

}

}

First, we're importing FormBuilder, FormGroup, Validators from @angular/forms.

Then we're setting a few boolean properties that will help us determine when the form has been submitted and if it validation is successful.

Then we're creating an instance of the formBuilder in the constructor. We then use this form building to construct our form properties in the ngOnInit() lifecycle hook.

We have two properties, name and message.

Then we created an onSubmit() method that will be called when the user submits the form. This is typically where you would call upon a method in the service to communicate with a mail service of sorts.

Next, visit contact.component.html:

<h1>Contact us</h1>

<form [formGroup]="messageForm" (ngSubmit)="onSubmit()">

<h5 \*ngIf="success">Your form is valid!</h5>

<label>

Name:

<input type="text" formControlName="name">

<div \*ngIf="submitted && messageForm.controls.name.errors" class="error">

<div \*ngIf="messageForm.controls.name.errors.required">Your name is required</div>

</div>

</label>

<label>

Message:

<textarea formControlName="message"></textarea>

<div \*ngIf="submitted && messageForm.controls.message.errors" class="error">

<div \*ngIf="messageForm.controls.message.errors.required">A message is required</div>

</div>

</label>

<input type="submit" value="Send message" class="cta">

</form>

<div \*ngIf="submitted" class="results">

<strong>Name:</strong>

<span>{{ messageForm.controls.name.value }}</span>

<strong>Message:</strong>

<span>{{ messageForm.controls.message.value }}</span>

</div>

Baked in here is a full form with validation. It also prints out the form values beneath it when the form has been submitted.

Let's update the css for the component to make it look decent:

label {

display: block;

input, textarea {

display: block;

width: 50%;

margin-bottom: 20px;

padding: 1em;

}

.error {

margin-top: -20px;

background: yellow;

padding: .5em;

display: inline-block;

font-size: .9em;

margin-bottom: 20px;

}

}

.cta {

background: #7700FF;

border: none;

color: white;

text-transform: uppercase;

border-radius: 4px;

padding: 1em;

cursor: pointer;

font-family: 'Montserrat';

}

.results {

margin-top: 50px;

strong {

display: block;

}

span {

margin-bottom: 20px;

display: block;

}

}

Save it, and the result in the browser should look like this!

