Beginning

In your command line, type:

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

Once complete, you can now access the CLI by simply starting any commands with ng . Hop into whichever folder you want to store your projects, and run the following command to install a new Angular project.

WARNING! Using git bash terminal caused unexpected behaviour so better to use cmd or another default terminal.

```
ng new playground
```

It's going to present you with a couple questions before beginning:

```
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS   [
http://sass-lang.com   ]
```

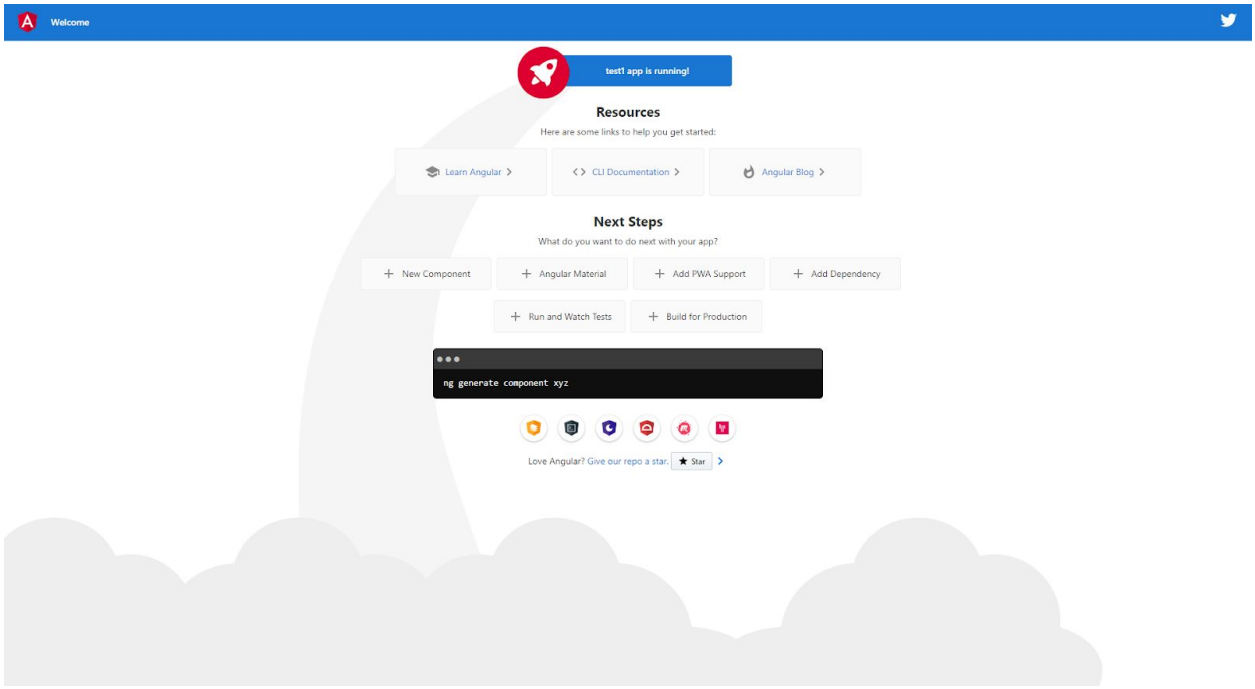
It will take a minute or two and once completed, you can now hop into the new project folder by typing:

```
cd playground
```

Open up this project in your preferred code editor (Visual Studio Code, for example), and then execute this command to run a development server using the Angular CLI:

```
ng serve -o
```

As a result the page should be opened on localhost:4200 in a browser:



What we have in the project?

- package.json - dependencies
- index.html - main page
- styles.css - common styles
- app.component - 4 files, root component
- app.module.ts - root module
- app-routing.module.ts - routing module (check if it present, recreate project if not)

Let's generate more components:

```
ng generate component nav
ng g c about
ng g c contact
ng g c home
```

Note that components are already registered in the root module. If you create components manually you have to do it by yourself.
Change your root (**app.component.html**) markup:

```
<app-nav></app-nav>
```

```
<section>
  <router-outlet></router-outlet>
</section>
```

Nav.component.html markup:

```
<header>
  <div class="container">
      <a routerLink="/" class="logo">Playground</a>
// router link is an attribute for router
// we don't use href here
      <nav>
      <ul>
      <li><a routerLink="/">Home</a></li>
      <li><a routerLink="/about">About</a></li>
      <li><a routerLink="/contact">Contact us</a></li>
      </ul>
      </nav>
  </div>
</header>
```

Styling

First, let's visit the global stylesheet by opening **/src/styles.scss** and define the following rulesets:

```
@import url('https://fonts.googleapis.com/css?family=Montserrat:400,700');
body, html {
    height: 100%;
    margin: 0 auto;
}
body {
    font-family: 'Montserrat';
    font-size: 18px;
}
a {
    text-decoration: none;
}
.container {
    width: 80%;
    margin: 0 auto;
```

```
    padding: 1.3em;
    display: grid;
    grid-template-columns: 30% auto;
    a {
      color: white;
    }
}
section {
    width: 80%;
    margin: 0 auto;
    padding: 2em;
}
```

Visit **nav/component.scss** and paste the following contents:

```
header {
    background: #7700FF;
    .logo {
      font-weight: bold;
    }
    nav {
      justify-self: right;

      ul {
            list-style-type: none;
            margin: 0; padding: 0;
            li {
                  float: left;
                  a {
                  padding: 1.5em;
                  text-transform: uppercase;
                  font-size: .8em;
                  &:hover {
                        background: #8E2BFF;
                  }
                  }
            }
      }
    }
}
```

Templating

What if we wanted to display properties that are coming from our component? We use string interpolation.
Make the following adjustment to our template:

Change:

```
<a routerLink="/">Playground</a>
```

To:

```
<a routerLink="/">{{ appTitle }}</a>
```

Let's define that property in **nav.component.ts**:

```
export class NavComponent implements OnInit {
  appTitle = 'Playground';
  constructor() { }
  ngOnInit() {
  }
}
```

More about interpolation https://angular.io/guide/template-syntax#interpolation-

Let's check the result. Save and open localhost:4200:

Routing

The Angular **Router** enables navigation from one view to the next as users perform application tasks. Let's register already added to markup **routerLink** s in **app-routing.module.ts** :

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
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 },
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

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.

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 and click on the button. The output should show clicked. Great!
You can experiment with the other event types by replacing **(click)** with the names below:

```
(focus)="myMethod()"
(blur)="myMethod()"
(submit)="myMethod()"
(scroll)="myMethod()"
(cut)="myMethod()"
(copy)="myMethod()"
(paste)="myMethod()"
(keydown)="myMethod()"
(keypress)="myMethod()"
(keyup)="myMethod()"
(mouseenter)="myMethod()"
(mousedown)="myMethod()"
(mouseup)="myMethod()"
(click)="myMethod()"
(dblclick)="myMethod()"
(drag)="myMethod()"
(dragover)="myMethod()"
(drop)="myMethod()"
```

Class and Style Binding

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 scss file:

```
.gray {
      color: gray;
}
```

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.

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>
```

Services

Services in Angular 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. Lets execute in terminal the following command to init a **service** with name **data**:

```
ng generate service data
```

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

```
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 app.

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';
```

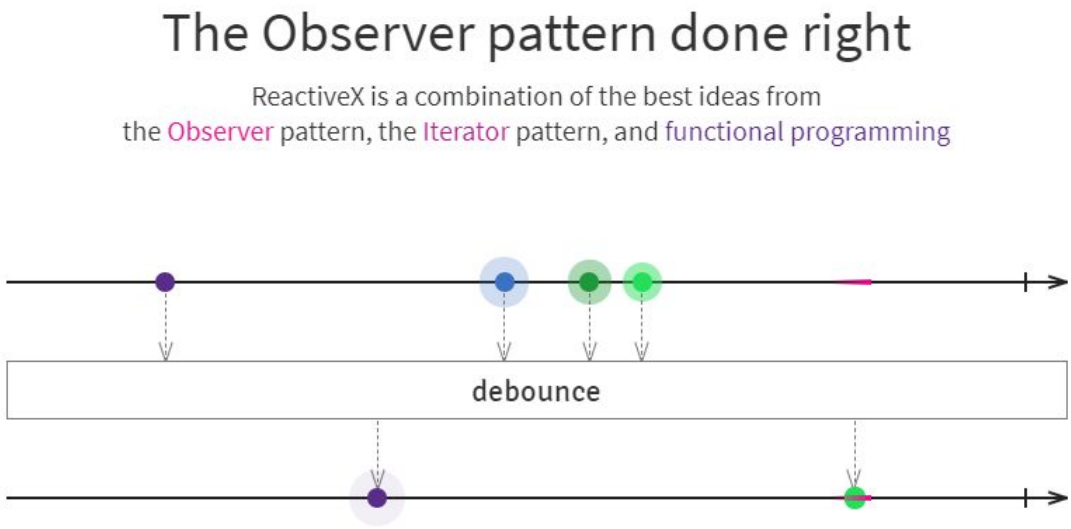
Next, add it to the imports array:

```
imports: [
      BrowserModule,
      AppRoutingModule,
      HttpClientModule, // <-- Right here
  ],
```

Now we can use it in our **/src/app/data.service.ts** file. Let's add an instance of HttpClient to constructor and also implement method **getUsers()**:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { UserData } from './response';
@Injectable({
  providedIn: 'root'

})
export class DataService {
  constructor(private http: HttpClient) { }

  getUsers(): Observable<Response> {
    return this.http.get<Response>('https://reqres.in/api/users');
  }

}
```

What have we done?
We use **http** variable that have been assigned an instance of HttpClient for retrieving data through **get()** method from **reqres.in** . This is a free public API that we can use to grab data. get() method will return Observable.
What is **Observable?**
Observable is nothing but a stream of data separated over time. Observable is a part of RxJs library. You can subscribe to this stream of data and do what you need with this data.



Observables is a new primitive type which acts as a blueprint for how we want to create streams, subscribe to them, react to new values, and combine streams together to build new ones.

RxJS is a library for composing asynchronous and event-based programs by using observable sequences. It provides one core type, the **Observable**, satellite types (Observer, Schedulers, Subjects) and operators inspired by Array#extras (map, filter, reduce, every, etc) to allow handling asynchronous events as collections. Many Angular APIs, including HttpClient, produce and consume RxJS Observables.
When we use http#get() we receive Observable, a stream of data that consists of mock User data in our case. We should **subscribe** to it to have a chance to get data from Observable. Without a **subscription**, Observer cannot receive any packages(data) from Observable. As we use **subscribe()** we can do anything with data chunks from **Observable. This is not the same as Java Streams but the idea of a data stream is pretty similar.**

Observable is a blank stream but to define a type of its data we should add a new type. Create new interfaces with names **response.ts (will represent the whole response on GET method of http instance) and user.ts (will represent a particular part of the response)** and fill them by this way:

user.ts

```
export interface User {
    first_name: string;
    last_name: string;
    avatar: string;
}
```

reponse.ts

```
import { User } from './user.model';

export class Response {
    data: User[];
}
```

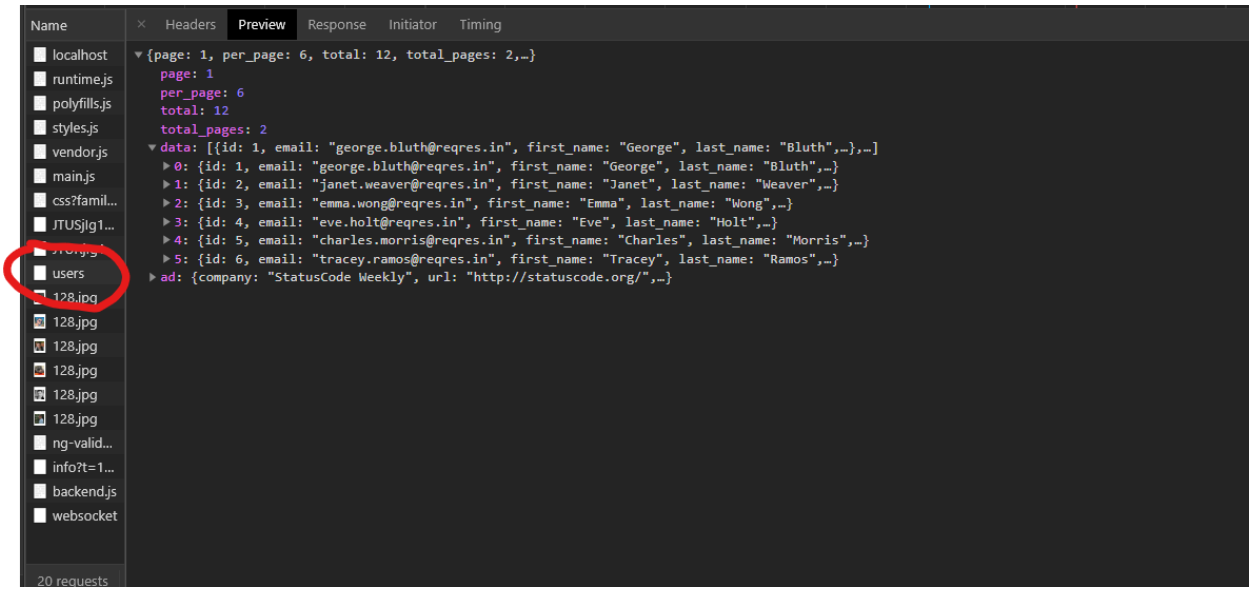
After implementation of this model interfaces don't forget to add import to service class if your IDE didn't auto-import it.

Now **service data#getUsers()** returns an **Observable** object and we should subscribe to it to retrieve data. Let's do that from **ngOnInit()** method of home.component.ts (ngOnInit method is a part of the component lifecycle, code in his body will be executed during initialization of a component. More about here https://angular.io/guide/lifecycle-hooks).

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

```
import { Component, OnInit } from '@angular/core';
import { DataService } from '../data.service';
import { Response } from '../response';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit {

  response: Response;

  constructor(private data: DataService) { }

  ngOnInit(): void {
    this.data.getUsers().subscribe((item: Response) => {
      this.response = item;
    });
  }

}
```

We're defining a response property, and then we're calling the data#getUsers() method and subscribing to it. You can check internals of response from https://reqres.in/api/users through chrome dev tools in the network tab:



Directives

Finish line: we should bind the data from https://reqres.in to view - html template **home.component.html.**

What are directives? This is just an instruction for the DOM. Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.

As with other directives, you apply a structural directive to a host element. The directive then does whatever it's supposed to do with that host element and its descendants. Structural directives are easy to recognize. An asterisk (*) precedes the directive attribute name as in this example.

Example:

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

***ngIf** defines whether a user sees an element of the DOM's structure or not depending on the boolean value. If the "hero" value is false the element will be hidden and vice versa.

***ngFor** helps to create a loop of similar elements in DOM. For example, in <ul> tag.

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

```
<h1>Users</h1>
<ul *ngIf="response">
    <li *ngFor="let user of response.data">
        <img [src]="user.avatar">
        <p>{{ user.first_name }} {{ user.last_name }}</p>
    </li>
</ul>
```

Also let's add some styles to **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%;
        }
    }
}
```

Save all changes and open localhost:4200 to check the result:

That's all. For more detailed information check documentation angular.io. Good Luck!