Angular

Table of Contents

About

Chapter 1: Getting started with Angular

Remarks

Angular (commonly referred to as "**Angular 2+**" or "**Angular 2**") is a TypeScript-based open-source front-end web framework led by the Angular Team at Google and by a community of individuals and corporations to address all of the parts of the developer's workflow while building complex web applications. Angular is a complete rewrite from the same team that built AngularJS. [1]

The framework consists of several libraries, some of them core (@angular/core for example) and some optional (@angular/animations).

You write Angular applications by composing HTML *templates* with Angularized markup, writing *component* classes to manage those templates, adding application logic in *services*, and boxing components and services in *modules*.

Then you launch the app by *bootstrapping* the *root module*. Angular takes over, presenting your application content in a browser and responding to user interactions according to the instructions you've provided.

Arguably, the most fundamental part of developing Angular applications are the **components**. A component is the combination of an HTML template and a component class that controls a portion of the screen. Here is an example of a component that displays a simple string:

src/app/app.component.ts

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

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

Every component begins with a `@Component` decorator function that takes a metadata object. The metadata object describes how the HTML template and component class work together.

The `selector` property tells Angular to display the component inside a custom `<my-app>` tag in the *index.html* file.

index.html (inside the `body` tag)

```
<my-app>Loading AppComponent content here ...</my-app>
```

The template property defines a message inside a `<h1>` header. The message starts with "Hello" and ends with `{{name}}`, which is an Angular interpolation binding expression. At runtime, Angular replaces `{{name}}` with the value of the component's `name` property. Interpolation binding is one of many Angular features you'll discover in this documentation. In the example, change the component class's `name` property from `'Angular'` to `'World'` and see what happens.

This example is written in **TypeScript**, a superset of JavaScript. Angular uses TypeScript because its types make it easy to support developer productivity with tooling. Additionally, **almost all support is for TypeScript** and so using **plain JavaScript** to write your application will be **difficult** . Writing Angular code in JavaScript is possible, however; this guide explains how.

More information on the **architecture** of Angular can be found **here**

Versions

Version	Release Date
5.0.0-beta.1 (Latest)	2017-07-27
4.3.2	2017-07-26
5.0.0-beta.0	2017-07-19
4.3.1	2017-07-19
4.3.0	2017-07-14
4.2.6	2017-07-08
4.2.5	2017-06-09
4.2.4	2017-06-21
4.2.3	2017-06-16
4.2.2	2017-06-12
4.2.1	2017-06-09
4.2.0	2017-06-08
4.2.0-rc.2	2017-06-01
4.2.0-rc.1	2017-05-26
4.2.0-rc.0	2017-05-19
4.1.3	2017-05-17
4.1.2	2017-05-10

Version	Release Date
4.1.1	2017-05-04
4.1.0	2017-04-26
4.1.0-rc.0	2017-04-21
4.0.3	2017-04-21
4.0.2	2017-04-11
4.0.1	2017-03-29
4.0.0	2017-03-23
4.0.0-rc.6	2017-03-23
4.0.0-rc.5	2017-03-17
4.0.0-rc.4	2017-03-17
2.4.10	2017-03-17
4.0.0-rc.3	2017-03-10
2.4.9	2017-03-02
4.0.0-rc.2	2017-03-02
4.0.0-rc.1	2017-02-24
2.4.8	2017-02-18
2.4.7	2017-02-09
2.4.6	2017-02-03
2.4.5	2017-01-25
2.4.4	2017-01-19
2.4.3	2017-01-11
2.4.2	2017-01-06
2.4.1	2016-12-21
2.4.0	2016-12-20
2.3.1	2016-12-15

Version	Release Date
2.3.0	2016-12-07
2.3.0-rc.0	2016-11-30
2.2.4	2016-11-30
2.2.3	2016-11-23
2.2.2	2016-11-22
2.2.1	2016-11-17
2.2.0	2016-11-14
2.2.0-rc.0	2016-11-02
2.1.2	2016-10-27
2.1.1	2016-10-20
2.1.0	2016-10-12
2.1.0-rc.0	2016-10-05
2.0.2	2016-10-05
2.0.1	2016-09-23
2.0.0	2016-09-14
2.0.0-rc.7	2016-09-13
2.0.0-rc.6	2016-08-31
2.0.0-rc.5	2016-08-09
2.0.0-rc.4	2016-06-30
2.0.0-rc.3	2016-06-21
2.0.0-rc.2	2016-06-15
2.0.0-rc.1	2016-05-03
2.0.0-rc.0	2016-05-02

Examples

Installation of Angular using angular-cli

This example is a quick setup of Angular and how to generate a quick example project.

Prerequisites:

- Node.js 6.9.0 or greater.
- npm v3 or greater or yarn.
- Typings v1 or greater.

Open a terminal and run the commands one by one:

`npm install -g typings` or `yarn global add typings`

`npm install -g @angular/cli` or `yarn global add @angular/cli`

The first command installs the typings library globally (and adds the `typings` executable to PATH). The second installs **@angular/cli** globally, adding the executable `ng` to PATH.

To setup a new project

Navigate with the terminal to a folder where you want to set up the new project.

Run the commands:

```
ng new PROJECT_NAME
cd PROJECT_NAME
ng serve
```

That is it, you now have a simple example project made with Angular. You can now navigate to the link displayed in terminal and see what it is running.

To add to an existing project

Navigate to the root of your current project.

Run the command:

```
ng init
```

This will add the necessary scaffolding to your project. The files will be created in the current directory so be sure to run this in an empty directory.

Running The Project Locally

In order to see and interact with your application while it's running in the browser you must start a local development server hosting the files for your project.

```
ng serve
```

If the server started successfully it should display an address at which the server is running. Usually is this:

```
http://localhost:4200
```

Out of the box this local development server is hooked up with Hot Module Reloading, so any changes to the html, typescript, or css, will trigger the browser to be automatically reloaded (but can be disabled if desired).

Generating Components, Directives, Pipes and Services

The `ng generate <scaffold-type> <name>` (or simply `ng g <scaffold-type> <name>`) command allows you to automatically generate Angular components:

```
# The command below will generate a component in the folder you are currently at
ng generate component my-generated-component
# Using the alias (same outcome as above)
ng g component my-generated-component
# You can add --flat if you don't want to create new folder for a component
ng g component my-generated-component --flat
# You can add --spec false if you don't want a test file to be generated (my-generated-
component.spec.ts)
ng g component my-generated-component --spec false
```

There are several possible types of scaffolds angular-cli can generate:

Scaffold Type	Usage
Module	`ng g module my-new-module`
Component	`ng g component my-new-component`
Directive	`ng g directive my-new-directive`
Pipe	`ng g pipe my-new-pipe`
Service	`ng g service my-new-service`
Class	`ng g class my-new-class`
Interface	`ng g interface my-new-interface`

Scaffold Type	Usage
Enum	`ng g enum my-new-enum`

You can also replace the type name by its first letter. For example:

`ng g m my-new-module` to generate a new module or `ng g c my-new-component` to create a component.

Building/Bundling

When you are all finished building your Angular web app and you would like to install it on a web server like Apache Tomcat, all you need to do is run the build command either with or without the production flag set. Production will minifiy the code and optimize for a production setting.

```
ng build
```

or

```
ng build --prod
```

Then look in the projects root directory for a `/dist` folder, which contains the build.

If you'd like the benefits of a smaller production bundle, you can also use Ahead-of-Time template compilation, which removes the template compiler from the final build:

```
ng build --prod --aot
```

Unit Testing

Angular provides in-built unit testing, and every item created by angular-cli generates a basic unit test, that can be expended. The unit tests are written using jasmine, and executed through Karma. In order to start testing execute the following command:

```
ng test
```

This command will execute all the tests in the project, and will re-execute them every time a source file changes, whether it is a test or code from the application.

For more info also visit: angular-cli github page

Angular "Hello World" Program

Prerequisites:

Setting up the Development Environment

Before we get started, we have to setup our environment.

- Install Node.js and npm if they are not already on your machine.

  Verify that you are running at least node 6.9.x and npm 3.x.x by running node -v and npm -v in a terminal/console window. Older versions produce errors, but newer versions are fine.

- Install the Angular CLI globally using `npm install -g @angular/cli`.

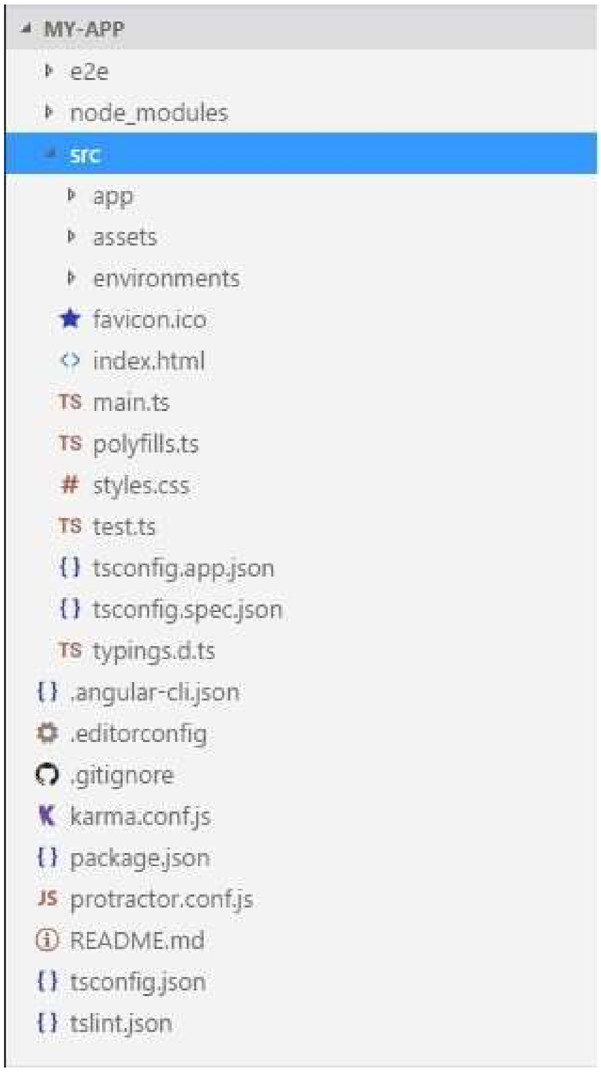
Step 1: Creating a new project

Open a terminal window (or Node.js command prompt in windows).

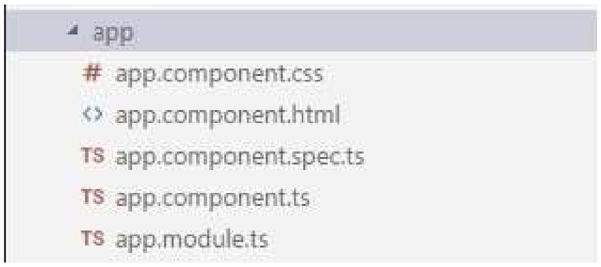
We create a new project and skeleton application using the command:

```
ng new my-app
```

Here the `ng` is for Angular. We get a file structure something like this.



```
MY-APP
  e2e
  node_modules
  src
    app
    assets
    environments
    favicon.ico
    index.html
    main.ts
    polyfills.ts
    styles.css
    test.ts
    tsconfig.app.json
    tsconfig.spec.json
    typings.d.ts
  .angular-cli.json
  .editorconfig
  .gitignore
  karma.conf.js
  package.json
  protractor.conf.js
  README.md
  tsconfig.json
  tslint.json
```



There are lots of files. We need not worry about all of them now.

Step 2: Serving the application

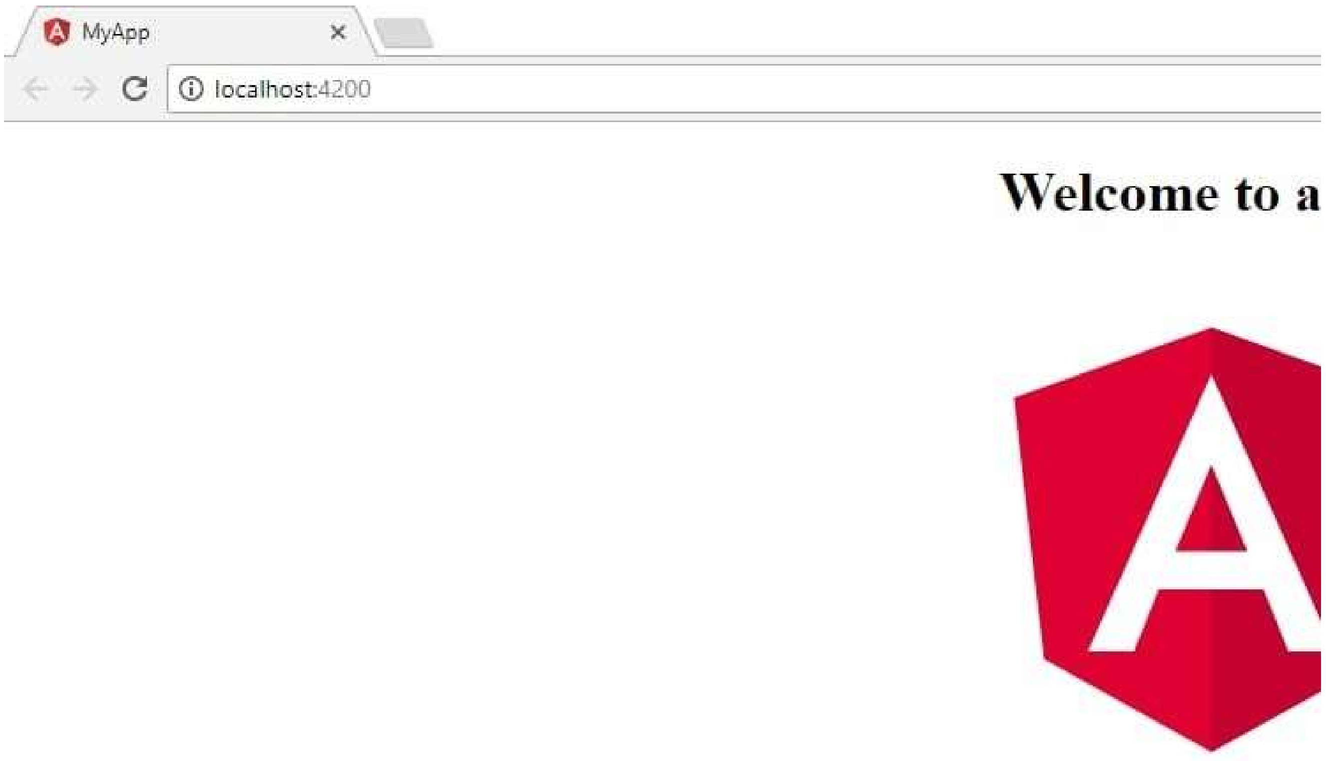
We launch our application using following command:

```
ng serve
```

We may use a flag `-open`(or simply `-o`) which will automatically open our browser on `http://localhost:4200/`

```
ng serve --open
```

Navigate browser to the address `http://localhost:4200/`. It looks something like this:

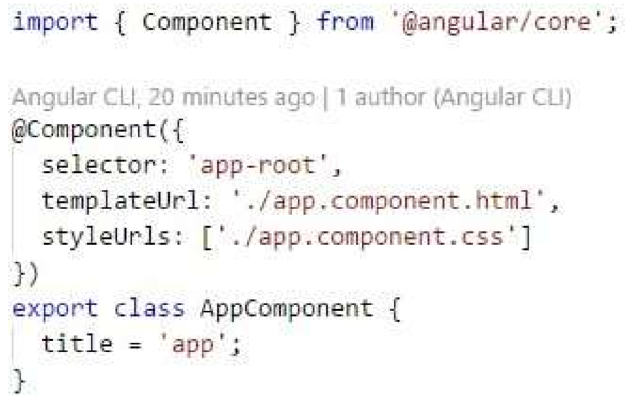


Step 3: Editing our first Angular component

The CLI created the default Angular component for us. This is the root component and it is named `app-root`. One can find it in `./src/app/app.component.ts`.

Open the component file and change the title property from `Welcome to app!!` to `Hello World`. The browser reloads automatically with the revised title.

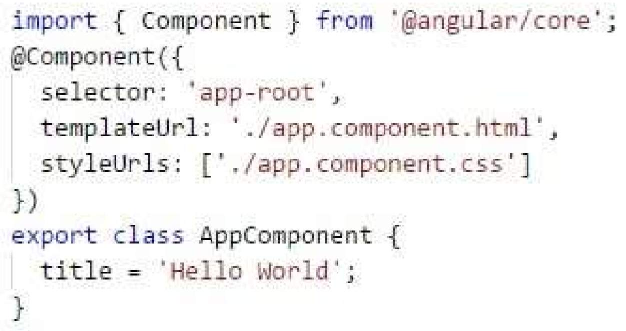
Original Code : Notice the `title = 'app';`



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

Angular CLI, 20 minutes ago | 1 author (Angular CLI)
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```

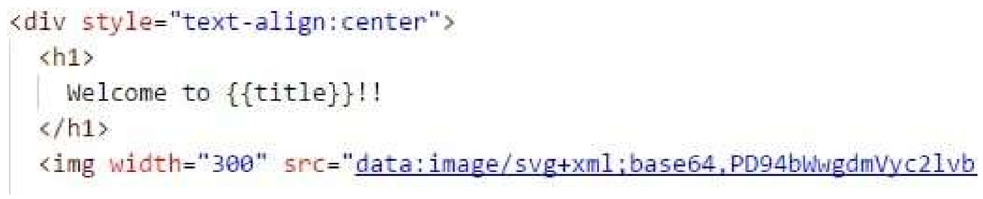
Modified Code : Value of `title` is changed.



```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Hello World';
}
```

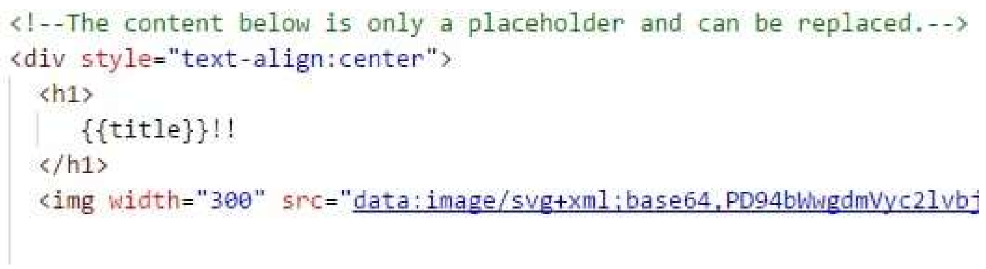
Similarly there is a change in `./src/app/app.component.html`.

Original HTML



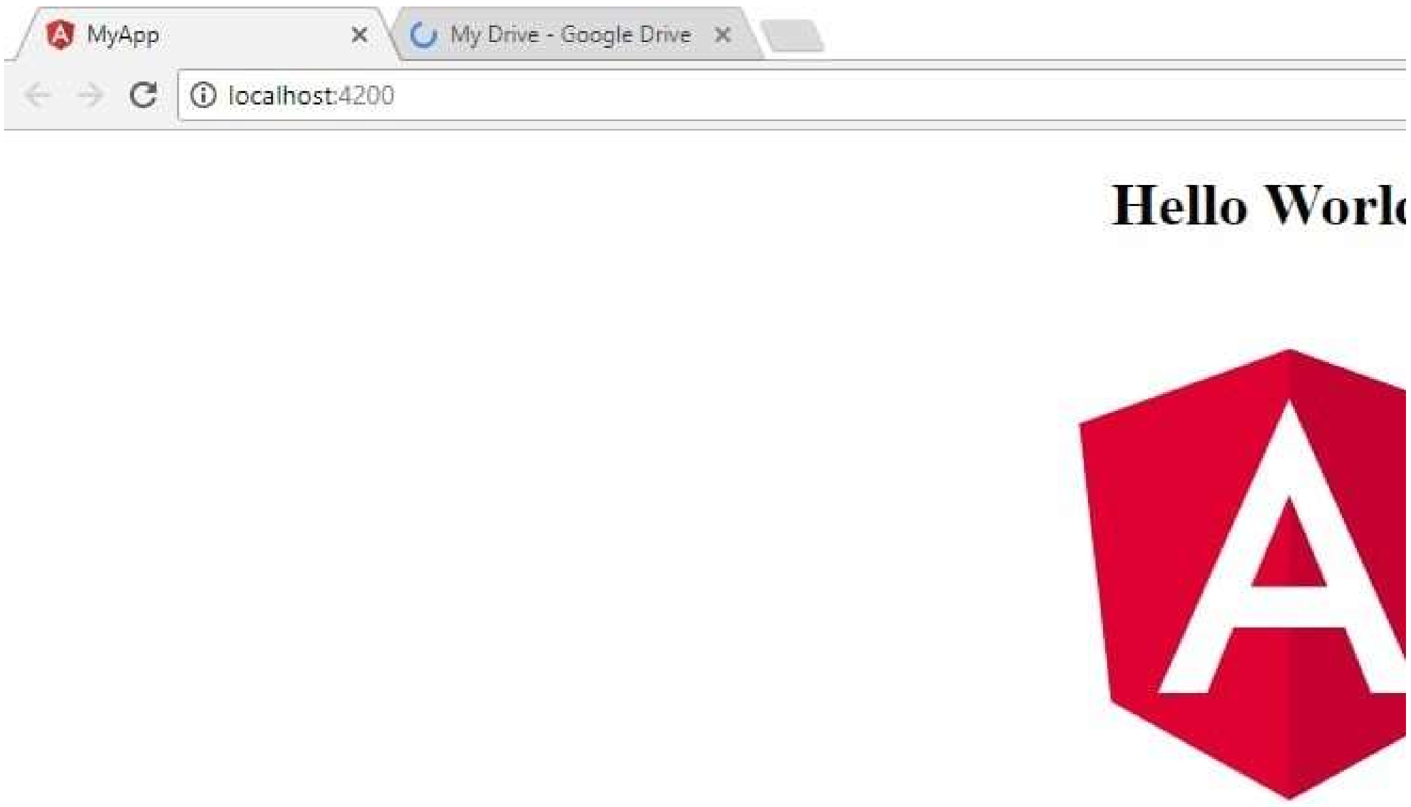
```
<div style="text-align:center">
  <h1>
    Welcome to {{title}}!!
  </h1>
  <img width="300" src="data:image/svg+xml;base64,PD94bWwgdmVyc2lvb
```

Modified HTML



```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    {{title}}!!
  </h1>
  <img width="300" src="data:image/svg+xml;base64,PD94bWwgdmVyc2lvbj
```

Notice that the value of `title` from the `./src/app/app.component.ts` will be displayed. The browser reloads automatically when the changes are done. It looks something like this.



Chapter 2: Event Emitter

Examples

Catching the event

Create a service-

```
import {EventEmitter} from 'angular2/core';
export class NavService {
    navchange: EventEmitter<number> = new EventEmitter();
    constructor() {}
    emitNavChangeEvent(number) {
        this.navchange.emit(number);
    }
    getNavChangeEmitter() {
        return this.navchange;
    }
}
```

Create a component to use the service-

```
import {Component} from 'angular2/core';
import {NavService} from '../services/NavService';

@Component({
    selector: 'obs-comp',
    template: `obs component, item: {{item}}`
    })
    export class ObservingComponent {
    item: number = 0;
    subscription: any;
    constructor(private navService:NavService) {}
    ngOnInit() {
        this.subscription = this.navService.getNavChangeEmitter()
        .subscribe(item => this.selectedNavItem(item));
    }
    selectedNavItem(item: number) {
        this.item = item;
    }
    ngOnDestroy() {
        this.subscription.unsubscribe();
    }
}

@Component({
    selector: 'my-nav',
    template:`
        <div class="nav-item" (click)="selectedNavItem(1)">nav 1 (click me)</div>
        <div class="nav-item" (click)="selectedNavItem(2)">nav 2 (click me)</div>
    `,
})
export class Navigation {
    item = 1;
    constructor(private navService:NavService) {}
    selectedNavItem(item: number) {
        console.log('selected nav item ' + item);
        this.navService.emitNavChangeEvent(item);
    }
}
```

Chapter 3: For Loop

Examples

NgFor - Markup For Loop

The **NgFor** directive instantiates a template once per item from an iterable. The context for each instantiated template inherits from the outer context with the given loop variable set to the current item from the iterable.

To customize the default tracking algorithm, NgFor supports **trackBy** option. **trackBy** takes a function which has two arguments: index and item. If **trackBy** is given, Angular tracks changes by the return value of the function.

```
<li *ngFor="let item of items; let i = index; trackBy: trackByFn">
    {{i}} - {{item.name}}
</li>
```

Additional Options: NgFor provides several exported values that can be aliased to local variables:

- **index** will be set to the current loop iteration for each template context.
- **first** will be set to a boolean value indicating whether the item is the first one in the iteration.
- **last** will be set to a boolean value indicating whether the item is the last one in the iteration.
- **even** will be set to a boolean value indicating whether this item has an even index.
- **odd** will be set to a boolean value indicating whether this item has an odd index.

Chapter 4: Forms

Examples

Reactive Forms

app.module.ts

Add these into your app.module.ts file to use reactive forms

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
    imports: [
        BrowserModule,
        FormsModule,
        ReactiveFormsModule,
    ],
    declarations: [ AppComponent ]
    providers: [],
    bootstrap: [ AppComponent ]
})
export class AppModule {}
```

app.component.ts

```
import { Component,OnInit } from '@angular/core';
import template from './app.component.html';
import { FormGroup,FormBuilder,Validators } from '@angular/forms';
import { matchingPasswords } from './validators';

@Component({
    selector: 'app',
    template
})
export class AppComponent implements OnInit {
    addForm: FormGroup;

    constructor(private formBuilder: FormBuilder) {
    }

    ngOnInit() {
        this.addForm = this.formBuilder.group({
            username: ['', Validators.required],
            email: ['', Validators.required],
            role: ['', Validators.required],
            password: ['', Validators.required],
            password2: ['', Validators.required]
        }, { validator: matchingPasswords('password', 'password2') });
    };

    addUser() {
        if (this.addForm.valid) {
            var adduser = {
                username: this.addForm.controls['username'].value,
                email: this.addForm.controls['email'].value,
                password: this.addForm.controls['password'].value,
                profile: {
                    role: this.addForm.controls['role'].value,
                    name: this.addForm.controls['username'].value,
                    email: this.addForm.controls['email'].value
                }
            };

            console.log(adduser);// adduser var contains all our form values. store it where
you want
            this.addForm.reset();// this will reset our form values to null
        }
    }

}
```

app.component.html

```
<div>
    <form [formGroup]="addForm">
        <input
            type="text"
            placeholder="Enter username"
            formControlName="username" />

        <input
            type="text"
            placeholder="Enter Email Address"
            formControlName="email"/>

        <input
            type="password"
            placeholder="Enter Password"
            formControlName="password" />

        <input
            type="password"
            placeholder="Confirm Password"
            name="password2"
            formControlName="password2" />

        <div class='error' *ngIf="addForm.controls.password2.touched">
            <div
                class="alert-danger errormessageadduser"
                *ngIf="addForm.hasError('mismatchedPasswords')">
                    Passwords do not match
            </div>
        </div>
        <select name="Role" formControlName="role">
            <option value="admin" >Admin</option>
            <option value="Accounts">Accounts</option>
            <option value="guest">Guest</option>
        </select>
        <br/>
        <br/>
        <button type="submit" (click)="addUser()">
            <span>
                <i class="fa fa-user-plus" aria-hidden="true"></i>
            </span>
            Add User
        </button>
    </form>
</div>
```

validators.ts

```
export function matchingPasswords(passwordKey: string, confirmPasswordKey: string) {
    return (group: ControlGroup): {
        [key: string]: any
    } => {
        let password = group.controls[passwordKey];
        let confirmPassword = group.controls[confirmPasswordKey];

        if (password.value !== confirmPassword.value) {
            return {
                mismatchedPasswords: true
            };
        }
    }
}
```

Template Driven Forms

Template - `signup.component.html`

```
<form #signUpForm="ngForm" (ngSubmit)="onSubmit()">

  <div class="title">
    Sign Up
  </div>

  <div class="input-field">
    <label for="username">username</label>
    <input
      type="text"
      pattern="\w{4,20}"
      name="username"
      required="required"
      [(ngModel)]="signUpRequest.username" />
  </div>

  <div class="input-field">
    <label for="email">email</label>
    <input
      type="email"
      pattern="^\S+@\S+$"
      name="email"
      required="required"
      [(ngModel)]="signUpRequest.email" />
  </div>

  <div class="input-field">
    <label for="password">password</label>
    <input
      type="password"
      pattern=".{6,30}"
      required="required"
      name="password"
      [(ngModel)]="signUpRequest.password" />
  </div>

  <div class="status">
    {{ status }}
  </div>

  <button [disabled]="!signUpForm.form.valid" type="submit">
    <span>Sign Up</span>
  </button>

</form>
```

Component - `signup.component.ts`

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

import { SignUpRequest } from './signup-request.model';

@Component({
  selector: 'app-signup',
  templateUrl: './signup.component.html',
  styleUrls: ['./signup.component.css']
})
export class SignupComponent {

  status: string;
  signUpRequest: SignUpRequest;

  constructor() {
    this.signUpRequest = new SignUpRequest();
  }

  onSubmit(value, valid) {
    this.status = `User ${this.signUpRequest.username} has successfully signed up`;
  }

}
```

Model - `signup-request.model.ts`

```
export class SignUpRequest {

  constructor(
    public username: string="",
    public email: string="",
    public password: string=""
  ) {}

}
```

App Module - `app.module.ts`

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

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

@NgModule({
  declarations: [
    AppComponent,
    SignupComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

App Component - `app.component.html`

```
<app-signup></app-signup>
```

Chapter 5: Pipes

Introduction

Pipes are very similar to filters in AngularJS in that they both help to transform the data into a specified format.The pipe character `|` is used to apply pipes in Angular.

Examples

Custom Pipes

my.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'myPipe'})
export class MyPipe implements PipeTransform {

  transform(value:any, args?: any):string {
    let transformedValue = value; // implement your transformation logic here
    return transformedValue;
  }

}
```

my.component.ts

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

@Component({
  selector: 'my-component',
  template: `{{ value | myPipe }}`
})
export class MyComponent {

    public value:any;

}
```

my.module.ts

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

import { MyComponent } from './my.component';
import { MyPipe } from './my.pipe';

@NgModule({
  imports: [
    BrowserModule,
  ],
  declarations: [
    MyComponent,
    MyPipe
  ],
})
export class MyModule { }
```

Multiple custom pipes

Having different pipes is a very common case, where each pipe does a different thing. Adding each pipe to each component may become a repetitive code.

It is possible to bundle all frequently used pipes in one `Module` and import that new module in any component needs the pipes.

breaklines.ts

```
import { Pipe } from '@angular/core';
/**
 * pipe to convert the \r\n into <br />
 */
@Pipe({ name: 'br' })
export class BreakLine {
    transform(value: string): string {
        return value == undefined ? value :
             value.replace(new RegExp('\r\n', 'g'), '<br />')
              .replace(new RegExp('\n', 'g'), '<br />');
    }
}
```

uppercase.ts

```
import { Pipe } from '@angular/core';
/**
 * pipe to uppercase a string
 */
@Pipe({ name: 'upper' })
export class Uppercase{
    transform(value: string): string {
        return value == undefined ? value : value.toUpperCase( );
    }
}
```

pipes.module.ts

```
import { NgModule } from '@angular/core';
import { BreakLine } from './breakLine';
import { Uppercase} from './uppercase';

@NgModule({
    declarations: [
        BreakLine,
        Uppercase
    ],
    imports: [
    ],
    exports: [
        BreakLine,
        Uppercase
    ]
    ,
})
export class PipesModule {}
```

my.component.ts

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

@Component({
  selector: 'my-component',
  template: `{{ value | upper | br}}`
})
export class MyComponent {

    public value: string;

}
```

my.module.ts

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

import { MyComponent } from './my.component';
import { PipesModule} from './pipes.module';

@NgModule({
  imports: [
    BrowserModule,
    PipesModule,
  ],
  declarations: [
    MyComponent,
  ],
})
```

Chapter 6: Routing

Examples

Routing with children

I found this to be the way to properly nest children routes inside the app.routing.ts or app.module.ts file (depending on your preference). This approach works when using either WebPack or SystemJS.

The example below shows routes for home, home/counter, and home/counter/fetch-data. The first and last routes being examples of redirects. Finally at the end of the example is a proper way to export the Route to be imported in a separate file. For ex. app.module.ts

To further explain, Angular requires that you have a pathless route in the children array that includes the parent component, to represent the parent route. It's a little confusing but if you think about a blank URL for a child route, it would essentially equal the same URL as the parent route.

```
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from "@angular/router";

import { HomeComponent } from "./components/home/home.component";
import { FetchDataComponent } from "./components/fetchdata/fetchdata.component";
import { CounterComponent } from "./components/counter/counter.component";

const appRoutes: Routes = [
    {
        path: "",
        redirectTo: "home",
        pathMatch: "full"
    },
    {
        path: "home",
        children: [
            {
                path: "",
                component: HomeComponent
            },
            {
                path: "counter",
                children: [
                    {
                        path: "",
                        component: CounterComponent
                    },
                    {
                        path: "fetch-data",
                        component: FetchDataComponent
                    }
                ]
            }
        ]
    },
    {
        path: "**",
        redirectTo: "home"
    }
];

@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes)
    ],
    exports: [
        RouterModule
    ]
})
export class AppRoutingModule { }
```

Great Example and Description via Siraj

Basic Routing

Router enables navigation from one view to another based on user interactions with the application.

Following are the steps in implementing basic routing in Angular -

NOTE: Ensure you have this tag:

```
<base href='/'>
```

as the first child under your head tag in your index.html file. This element states that your app folder is the application root. Angular would then know how to organize your links.

1. Check if you are pointing to the correct/latest routing dependencies in package.json (using the latest version of Angular) and that you already did an `npm install` -

   ```
   "dependencies": {
       "@angular/router": "^4.2.5"
   }
   ```

2. Define the route as per its interface definition:

   ```
   interface Route {
     path?: string;
     pathMatch?: string;
     component?: Type<any>;
   }
   ```

3. In a routing file (`routes/app.routing.ts`), import all the components which you need to configure for different routing paths. Empty path means that view is loaded by default. ":" in the path indicates dynamic parameter passed to the loaded component.

   ```
   import { Routes, RouterModule } from '@angular/router';
   import { ModuleWithProviders } from '@angular/core';
   ```

```
import { BarDetailComponent } from '../components/bar-detail.component';
import { DashboardComponent } from '../components/dashboard.component';
import { LoginComponent } from '../components/login.component';
import { SignupComponent } from '../components/signup.component';

export const APP_ROUTES: Routes = [
    { path: '', pathMatch: 'full', redirectTo: 'login' },
    { path: 'dashboard', component: DashboardComponent },
    { path: 'bars/:id', component: BarDetailComponent },
    { path: 'login', component: LoginComponent },
    { path: 'signup',   component: SignupComponent }
];
export const APP_ROUTING: ModuleWithProviders = RouterModule.forRoot(APP_ROUTES);
```

4. In your `app.module.ts`, place this under `@NgModule([])` under `imports`:

```
// Alternatively, just import 'APP_ROUTES
import {APP_ROUTING} from '../routes/app.routing.ts';
@NgModule([
    imports: [
        APP_ROUTING
        // Or RouterModule.forRoot(APP_ROUTES)
    ]
])
```

5. Load/display the router components based on path accessed. The `<router-outlet>`directive is used to tell angular where to load the component.

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

@Component({
    selector: 'demo-app',
    template: `
        <div>
            <router-outlet></router-outlet>
        </div>
    `
})
export class AppComponent {}
```

6. Link the other routes. By default, `RouterOutlet` will load the component for which empty path is specified in the `Routes`. `RouterLink` directive is used with html anchor tag to load the components attached to routes. `RouterLink` generates the href attribute which is used to generate links. For example:

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

@Component({
    selector: 'demo-app',
    template: `
        <a [routerLink]="['/login']">Login</a>
        <a [routerLink]="['/signup']">Signup</a>
        <a [routerLink]="['/dashboard']">Dashboard</a>
        <div>
            <router-outlet></router-outlet>
        </div>
        `
})
export class AnotherComponent { }
```

Now, we are good with routing to static paths. `RouterLink` supports dynamic path too by passing extra parameters along with the path.

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

@Component({
  selector: 'demo-app',
  template: `
        <ul>
          <li *ngFor="let bar of bars | async">
            <a [routerLink]="['/bars', bar.id]">
              {{bar.name}}
            </a>
          </li>
        </ul>
    <div>
      <router-outlet></router-outlet>
    </div>
  `
})
export class SecondComponent { }
```

`RouterLink` takes an array where the first parameter is the path for routing and subsequent elements are for the dynamic routing parameters.

Chapter 7: RXJS and Observables

Examples

Wait for multiple requests

One common scenario is to wait for a number of requests to finish before continuing. This can be accomplished using the `forkJoin` method.

In the following example, `forkJoin` is used to call two methods that return `Observables`. The callback specified in the `.subscribe` method will be called when both Observables complete. The parameters supplied by `.subscribe` match the order given in the call to `.forkJoin`. In this case, first `posts` then `tags`.

```
loadData() : void {
    Observable.forkJoin(
        this.blogApi.getPosts(),
        this.blogApi.getTags()
    ).subscribe((([posts, tags]: [Post[], Tag[]]) => {
        this.posts = posts;
        this.tags = tags;
    }));
}
```

Basic Request

The following example demonstrates a simple HTTP GET request. `http.get()` returns an `Observable` which has the method `subscribe`. This one appends the returned data to the `posts` array.

```
var posts = []

getPosts(http: Http): {
    this.http.get(`https://jsonplaceholder.typicode.com/posts`)
        .subscribe(response => {
            posts.push(response.json());
        });
}
```

Chapter 8: Sharing data among components

Introduction

The objective of this topic is to create simple examples of several ways data can be shared between components via data binding and shared service.

Remarks

There are always many of ways of accomplishing one task in programming. Please feel free to edit current examples or add some of your own.

Examples

Sending data from parent component to child via shared service

service.ts:

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

@Injectable()
export class AppState {

  public mylist = [];

}
```

parent.component.ts:

```
import {Component} from '@angular/core';
import { AppState } from './shared.service';

@Component({
  selector: 'parent-example',
  templateUrl: 'parent.component.html',
})
export class ParentComponent {
  mylistFromParent = [];

  constructor(private appState: AppState){
    this.appState.mylist;
  }

  add() {
    this.appState.mylist.push({"itemName":"Something"});
  }

}
```

parent.component.html:

```
<p> Parent </p>
  <button (click)="add()">Add</button>
<div>
  <child-component></child-component>
</div>
```

child.component.ts:

```
import {Component, Input } from '@angular/core';
import { AppState } from './shared.service';

@Component({
  selector: 'child-component',
  template: `
    <h3>Child powered by shared service</h3>
        {{mylist | json}}
  `,
})
export class ChildComponent {
  mylist: any;

  constructor(private appState: AppState){
    this.mylist = this.appState.mylist;
  }

}
```

Send data from parent component to child component via data binding using @Input

parent.component.ts:

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

@Component({
  selector: 'parent-example',
  templateUrl: 'parent.component.html',
})

export class ParentComponent {
  mylistFromParent = [];

  add() {
    this.mylistFromParent.push({"itemName":"Something"});
  }

}
```

parent.component.html:

```
<p> Parent </p>
  <button (click)="add()">Add</button>

<div>
  <child-component [mylistFromParent]="mylistFromParent"></child-component>
</div>
```

child.component.ts:

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

@Component({
  selector: 'child-component',
  template: `
    <h3>Child powered by parent</h3>
        {{mylistFromParent | json}}
  `,
})

export class ChildComponent {
  @Input() mylistFromParent = [];
}
```

Sending data from child to parent via @Output event emitter

event-emitter.component.ts

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

@Component({
  selector: 'event-emitting-child-component',
  template: `<div *ngFor="let item of data">
               <div (click)="select(item)">
                 {{item.id}} = {{ item.name}}
               </div>
             </div>
            `
})

export class EventEmitterChildComponent implements OnInit{

  data;

  @Output()
  selected: EventEmitter<string> = new EventEmitter<string>();

  ngOnInit(){
    this.data = [ { "id": 1, "name": "Guy Fawkes", "rate": 25 },
                  { "id": 2, "name": "Jeremy Corbyn", "rate": 20 },
                  { "id": 3, "name": "Jamie James", "rate": 12 },
                  { "id": 4, "name": "Phillip Wilson", "rate": 13 },
                  { "id": 5, "name": "Andrew Wilson", "rate": 30 },
                  { "id": 6, "name": "Adrian Bowles", "rate": 21 },
                  { "id": 7, "name": "Martha Paul", "rate": 19 },
                  { "id": 8, "name": "Lydia James", "rate": 14 },
                  { "id": 9, "name": "Amy Pond", "rate": 22 },
                  { "id": 10, "name": "Anthony Wade", "rate": 22 } ]
  }

  select(item) {
      this.selected.emit(item);

  }

}
```

event-receiver.component.ts:

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

@Component({
    selector: 'event-receiver-parent-component',
    template: `<event-emitting-child-component (selected)="itemSelected($event)">
                </event-emitting-child-component>
                <p *ngIf="val">Value selected</p>
                <p style="background: skyblue">{{ val | json}}</p>`
})

export class EventReceiverParentComponent{
  val;

  itemSelected(e){
    this.val = e;
  }
}
```

Sending data asynchronous from parent to child using Observable and Subject

shared.service.ts:

```
import { Injectable }    from '@angular/core';
import { Headers, Http } from '@angular/http';

import 'rxjs/add/operator/toPromise';

import { Observable } from 'rxjs/Observable';
import { Observable } from 'rxjs/Rx';
import {Subject} from 'rxjs/Subject';


@Injectable()
export class AppState {

  private headers = new Headers({'Content-Type': 'application/json'});
  private apiUrl = 'api/data';

  // Observable string source
  private dataStringSource = new Subject<string>();

  // Observable string stream
  dataString$ = this.dataStringSource.asObservable();

  constructor(private http: Http) { }

  public setData(value) {
    this.dataStringSource.next(value);
  }

  fetchFilterFields() {
    console.log(this.apiUrl);
    return this.http.get(this.apiUrl)
               .delay(2000)
               .toPromise()
```

```
                .then(response => response.json().data)
                .catch(this.handleError);
  }

  private handleError(error: any): Promise<any> {
    console.error('An error occurred', error); // for demo purposes only
    return Promise.reject(error.message || error);
  }

}
```

parent.component.ts:

```
import {Component, OnInit} from '@angular/core';
import 'rxjs/add/operator/toPromise';
import { AppState } from './shared.service';

@Component({
  selector: 'parent-component',
  template: `
              <h2> Parent </h2>
              <h4>{{promiseMarker}}</h4>

              <div>
                <child-component></child-component>
              </div>
            `
})
export class ParentComponent implements OnInit {

  promiseMarker = "";

  constructor(private appState: AppState){ }

  ngOnInit(){
    this.getData();
  }

  getData(): void {
    this.appState
        .fetchFilterFields()
        .then(data => {
          // console.log(data)
          this.appState.setData(data);
          this.promiseMarker = "Promise has sent Data!";
        });
  }

}
```

child.component.ts:

```
import {Component, Input } from '@angular/core';
import { AppState } from './shared.service';


@Component({
  selector: 'child-component',
  template: `
```

```
    <h3>Child powered by shared service</h3>
        {{fields | json}}
  `,
})
export class ChildComponent {
  fields: any;

  constructor(private appState: AppState){
    // this.mylist = this.appState.get('mylist');

    this.appState.dataString$.subscribe(
      data => {
        // console.log("Subs to child" + data);
        this.fields = data;
      });

  }

}
```