Angular - A Beginner's Guide

This is a beginner's guide to Angular. It's a collection of notes and code snippets from the Angular Crash Course by Traversy Media and the official Angular documentation.

What is Angular?

- A frontend Javascript framework
- Created by Google
- Used to build Single Page Applications (SPAs)
- Uses TypeScript

It typically runs on the client side, in the browser but can also be used on the server through Node.js or other server-side frameworks.

Angular vs AngularJS

AngularJS is the first version of Angular. It was released in 2010 and is also known as Angular 1. It's still used in many legacy projects. Angular 2 was released in 2016 and is a complete rewrite of AngularJS. It's a completely different framework. Angular 2+ is just called Angular. Angular 2+ is much more performant and has a lot more features. It's also much easier to learn and use. AngularJS is no longer supported by Google.

Why Angular?

- Full featured framework (routing, forms, validation, http, etc)
- Proffered and popular in the enterprise world
- TypeScript support (optional)
- Test friendly (unit, e2e, etc)
- CLI (scaffolding, build, etc)

Prerequisites

- HTML, CSS, JS
- TypeScript (optional)
- OOP (classes, objects, interfaces, etc)
- Node.js and NPM
- Asynchronous programming (promises, observables, etc)
- Array methods (forEach, map, filter, reduce, etc)
- Fetch API and HTTP (GET, POST, PUT, DELETE, etc)

Angular CLI

- Standard tooling for Angular development.
- Create new projects, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.
- Dev server and easy production builds.
- Generate components, services, pipes, directives, etc.

Install Angular CLI globally

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

Create new project

```
ng new my-app
```

Run dev server

```
ng serve
```

The app will be available at http://localhost:4200

Angular Components

- Components are the fundamental building blocks of Angular applications.
- They display data on the screen, listen for user input, and take action based on that input.
- A component is a TypeScript class with an HTML template and an optional style sheet.
- Used to break up the application into smaller pieces of reusable code.
- Are reusable and can be used in other components.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  services: [PostsService],
})
export class AppComponent {
  /** Properties */
  title = 'my-app';
}
```

- The `@Component` decorator specifies the Angular metadata for the component.
- The `selector` property defines the HTML tag that represents the component (parent component).
- The `templateUrl` property defines the HTML template for the component.
- The `styleUrls` property defines the CSS styles for the component.
- The `services` property defines the services used by the component.

First Component

app.component.ts

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

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

app.component.html

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

<!-- Run any js code -->
{{2 + 5}}
```

Output

```
Welcome to My World
7
```

Create a new component

```
ng generate component components/navbar
```

This will create a new folder called `navbar` with the following files:

- `navbar.component.ts`
- `navbar.component.html`
- `navbar.component.css`
- `navbar.component.spec.ts`

Now we can use the `navbar` component in the `app` component.

app.component.html

```
<main>
  <app-navbar></app-navbar>
</main>
```

Passing data to a component

navbar.component.html

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

  <!-- Nav Button Component  -->
  <app-button color="green" text="Add"></app-button>
</nav>
```

navbar.component.ts

```
export class ButtonComponent {
  @Input() text: string;
  @Input() color: string;
}
```

button.component.html

```
<button [ngStyle]="{'background': color}" class="btn">{{text}}</button>

<!-- [ngStyle] is a directive -->
```

Adding Events to a Component

button.component.html

```
<button (click)="onClick()" class="btn">{{text}}</button>
```

button.component.ts

```
export class ButtonComponent {
  @Input() text: string;
  @Output() btnClick = new EventEmitter();

  onClick() {
    this.btnClick.emit();
  }
}
```

The `btnClick` event will be emitted when the button is clicked. We can listen for this event in the `navbar` component or any other component that uses the `button` component.

Every button click works differently. We can pass a function to the `btnClick` event (our custom event) and execute it when the button is clicked.

navbar.component.html

```
<app-button (btnClick)="addTask()" text="Add"></app-button>
```

navbar.component.ts

```
export class NavbarComponent {
  addTask() {
    console.log('Added Task');
  }
}
```

Looping through list of Items

Case 1: Looping in same component

app.component.ts

```
export class AppComponent {
  title = 'my-app';
  name = 'John Doe';
  age = 30;
  address = {
    street: '50 Main st',
    city: 'Boston',
    state: 'MA',
  };
  hobbies = ['music', 'movies', 'sports'];
}
```

app.component.html

```
<h1>{{title}}</h1>
<p>My name is {{name}} and I am {{age}} years old</p>
<p>My address is {{address.street}}, {{address.city}}, {{address.state}}</p>
<p>My hobbies are:</p>
<ul>
  <li *ngFor="let hobby of hobbies">{{hobby}}</li>
  <!-- *ngFor is a directive -->
</ul>
```

Case 2: Looping in a child component by passing array as input

app.component.ts

```
export class AppComponent {
  title = 'my-app';
  hobbies = ['music', 'movies', 'sports'];
}
```

app.component.html (Parent)

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

<app-navbar [hobbies]="hobbies"></app-navbar>
```

navbar.component.ts

```
export class NavbarComponent {
  @Input() hobbies: string[];
}
```

navbar.component.html (Child)

```
<ul>
  <li *ngFor="let hobby of hobbies">{{hobby}}</li>
</ul>
```

Case 3: Looping in a child component by passing single array item as input

app.component.ts

```
export class AppComponent {
  title = 'my-app';
  hobbies = ['music', 'movies', 'sports'];
}
```

app.component.html (Parent)

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

<app-navbar *ngFor="let hobby of hobbies" [hobby]="hobby"></app-navbar>
```

navbar.component.ts

```
export class NavbarComponent {
  @Input() hobby: string;
}
```

navbar.component.html (Child)

```
<li>{{hobby}}</li>
```

Angular Font Awesome Integration

There are multiple ways to integrate Font Awesome in Angular.

Method 1: Using `ng add` command

```
ng add @fortawesome/angular-fontawesome
```

Method 2: Using `npm` command

```
npm install @fortawesome/fontawesome-svg-core
npm install @fortawesome/free-solid-svg-icons
npm install @fortawesome/angular-fontawesome
```

Now to use the icons in our components, we need to import the `FontAwesomeModule` in the `app.module.ts` file.

app.module.ts

```
import { FontAwesomeModule } from '@fortawesome/angular-fontawesome';

@NgModule({
  imports: [FontAwesomeModule],
})
export class AppModule {}
```

Now we can use the icons in our components.

app.component.html

```
<fa-icon icon="coffee"></fa-icon>
```

We can also use the icons in our components by importing the `faCoffee` icon.

app.component.ts

```
import { faCoffee } from '@fortawesome/free-solid-svg-icons';

export class AppComponent {
  faCoffee = faCoffee;
}
```

app.component.html

```
<fa-icon [icon]="faCoffee"></fa-icon>
```

Angular Services

- Increase modularity and reusability of code.
- Components can give certain tasks to services and then listen for the result.
- These tasks can be anything such as fetching data from a server, logging data, or validating user input.
- This makes components lean and focused on supporting the view, and makes it easy to unit-test components with a mock service.

Create a new service

```
ng generate service services/task
```

This will create a new folder called `services/task` with the following files:

- `task.service.ts`
- `task.service.spec.ts`

Using a service into a component

task.service.ts

```
import { Injectable } from '@angular/core';
import { Task } from 'src/app/interfaces/Task';
import { TASKS } from 'src/app/data/mock-tasks';

@Injectable({
  providedIn: 'root',
})
export class TaskService {
  constructor() {}

  getTasks(): Task[] {
    return TASKS;
  }
}
```

app.component.ts

```
import { TaskService } from './services/task.service';

export class AppComponent {
  tasks: Task[] = [];

  constructor(private taskService: TaskService) {}

  ngOnInit(): void {
    this.tasks = this.taskService.getTasks();
  }
}
```

Observables and RxJS

- Observables are lazy. They don't do anything until something subscribes to them.
- Observables are able to deliver values either synchronously or asynchronously.
- Observables are cancelable. When an observer is no longer interested in an Observable, they can unsubscribe and the Observable will stop emitting items.

Implementing Observables

task.service.ts

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Task } from 'src/app/interfaces/Task';
import { TASKS } from 'src/app/data/mock-tasks';

export class TaskService {
  constructor() {}

  getTasks(): Observable<Task[]> {
    const tasks = of(TASKS); // of() converts the TASKS array into an Observable
    return tasks;
  }
}
```

app.component.ts

```
import { TaskService } from './services/task.service';

export class AppComponent {
  tasks: Task[] = [];

  constructor(private taskService: TaskService) {}

  ngOnInit(): void {
    this.taskService.getTasks().subscribe((tasks) => (this.tasks = tasks));
  }
}
```

Here we are subscribing to the `getTasks()` method of the `TaskService` and assigning the result to the `tasks` property of the `AppComponent`. This is an asynchronous operation.

It works similar to the `Promise` object and handling them using `then()` method.

Angular HTTP Client

Angular provides a simplified client HTTP API for Angular applications, the `HttpClientModule` which is based on `XMLHttpRequest` interface exposed by browsers.

Importing HttpClientModule

In order to use the `HttpClient` service, we need to import it as a module in the `app.module.ts` file.

app.module.ts

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

@NgModule({
  imports: [HttpClientModule],
})
export class AppModule {}
```

Using the HttpClient service

We need to pass the `HttpClient` service as a parameter to the constructor because Angular uses a mechanism called **Dependency Injection** to inject the `HttpClient` service into the `TaskService` service.

This is similar to the way we injected the `TaskService` service into the `AppComponent`.

task.service.ts

```
import { Observable } from 'rxjs';
import { Task } from 'src/app/interfaces/Task';
import { HttpClient, HttpHeaders } from '@angular/common/http';

export class TaskService {
  private apiUrl = 'http://localhost:5000/tasks';

  constructor(private http: HttpClient) {}

  getTasks(): Observable<Task[]> {
    // GET request
    const tasks = this.http.get<Task[]>(this.apiUrl);
    return tasks;
  }
}
```

This will return an `Observable` of type `Task[]` which we can subscribe to in the `AppComponent`.

Forms in Angular

Before we can use the `ngModel` directive, we need to import the `FormsModule` in the `app.module.ts` file.

app.module.ts

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

@NgModule({
  imports: [FormsModule],
})
export class AppModule {}
```

Two-way data binding

Create a two-way data binding using `ngModel` directive.

app.component.html

```
<input type="text" [(ngModel)]="title" />
```

app.component.ts

```
export class AppComponent {
  title: string = '';
}
```

Form Submission

We can use the `ngSubmit` directive to submit the form. This will call the `onSubmit()` method of the `AppComponent` when the form is submitted. We also don't need to do `preventDefault()` here.

app.component.html

```
<form (ngSubmit)="onSubmit()">
  <input type="text" [(ngModel)]="title" name="title" />
  <input type="submit" value="Save" />
</form>
```

app.component.ts

```
export class AppComponent {
  title: string = '';

  onSubmit() {
    console.log(this.title);
  }
}
```

Routing in Angular

Although we have the option to get the router setup by default when we create a new project using the `--routing` flag, we can also add it later.

Import `RouterModule` to `app.module.ts`

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

Add routes to `app.module.ts`

```
const appRoutes: Routes = [
  { path: '', component: TasksComponent },
  { path: 'about', component: AboutComponent },
];
```

Add `RouterModule` to `app.module.ts`

```
imports: [BrowserModule, HttpClientModule, FormsModule, RouterModule.forRoot(appRoutes)],
```

Add `router-outlet` to `app.component.html`

```
<div class="container">
  <app-header></app-header>
  <router-outlet></router-outlet>
  <app-footer></app-footer>
</div>
```

Add `routerLink` to `header.component.html`

```
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/about">About</a>
</nav>
```

Conclusion

Thank you for viewing this cheat sheet!

If you found it helpful please check out more of my work on yodkwtf.com or follow me on twitter. I also run a small youtube channel called Yodkwtf Academy.