Angular
~~Angular 2~~ Deep Dive

By Chris McKnight

Angular Cli

The Angular CLI is a command line interface tool that can create a project, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

Why would you want to use the Angular cli?
Rapid development, Angular build setup is tedious

Angular Cli Installation

```
npm i -g @angular/cli
# yarn global add @angular/cli
```

Angular Cli allows developers to rapidly develop applications with minimal setup

Create a project

```
ng new --link-cli --routing --style scss my-project
```

```
├── e2e
├── package.json
├── protractor.conf.js
├── src
│   ├── app
│   │   ├── app-routing.module.ts
│   │   ├── app.component.ts
│   │   └── app.module.ts
│   ├── index.html
│   ├── main.ts
│   ├── polyfills.ts
│   ├── styles.scss
├── tsconfig.json
└── yarn.lock
```

Global defaults can be set using `ng set --global` or by manually editing `~/.angular-cli.json`

Example: `ng set --global defaults.styleExt=scss`

Angular Cli common commands

- new
- serve (server)
- build
- generate
- test
- e2e
- lint

Some commands are aliases
serve => s build => b generate => g test => t e2e => e lint => l

Angular

One framework.

Mobile & Desktop

Angular is a development platform for building mobile and desktop web applications.

Decorators

A function that adds metadata to a class, its members (properties, methods) and function arguments.

Angular uses decorators extensively to create `directives`, `services`, etc.

Decorators are an experimental (stage 2), JavaScript language feature. TypeScript adds support for decorators.

Also known as an `annotation`

Decorator example

```
import { deprecate } from 'core-decorators';
import { compact } from 'lodash';

class Person {
  ...

  // deprecate the name method using a decorator
  @deprecate('Use the fullName method instead')
  get name() {
    return compact([this.first_name, this.last_name]).join(' ');
  }

  get fullName() {
    ...
  }
}
```

Three types of Directives

- Attribute directives
- Structural directives
- Components

We will explore all 3 categories of `directives`

Attribute Directives

See https://angular.io/docs/ts/latest/guide/attribute-directives.html for more

A category of directive that can listen to and modify the behavior of other HTML elements, attributes, properties, and components. They are usually represented as HTML attributes, hence the name.

Example demonstrating the `ngClass` attribute directive and a custom `myHighlight` directive

```
<my-app>
  <div [ngClass]="{active: active}" [myHighlight]="color"></div>
</my-app>
```

Structural Directives

For more see https://angular.io/docs/ts/latest/guide/structural-directives.html

A category of directive that can shape or reshape HTML layout, typically by adding and removing elements in the DOM.

- ngIf
- ngFor
- ngSwitch

Structural Directive Example

The example shows the use of `ngFor` to generate an element, `li`, for each person's name.

```
<ul>
  <li *ngFor="let person of people">{{person.name}}</li>
</ul>
```

Components

- Fundamental building block to Angular
- Handles exposing data to view and handling user interaction
- A directive containing a template

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

@Component({
  selector: 'home',
  styles: [`
  h1 { color: rebeccapurple; }
  `],
  template: `
  <h1>{{title}}</h1>
  `
})
export default class HomeComponent {
  public title: string = 'Hello World';
}
```

The selector is the "tag" to use inside of another component's template. It is also how the component will appear in the browser.

Components can be split into 2 categories of components, containers (smart) and presentational (dumb).

Container components

a.k.a. "smart" components

- Root level, routable components
- Access application state from ngrx, services, etc
- Handle events emitted from child components within the same view tree

Presentational Components

a.k.a. ”dumb” components

- Small
- Focused
- Reusable
- State down, Events up

"State down, Events up" allow large applications to render more efficiently. Think flux architecture.

Overview of managing state with ngrx: http://onehungrymind.com/build-better-angular-2-application-redux-ngrx/

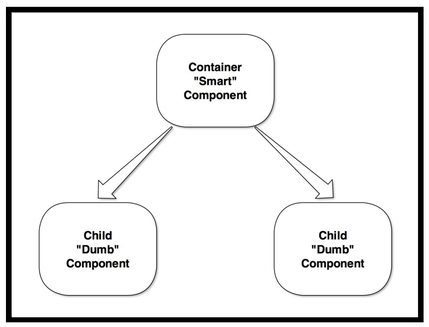
Presentational Component Example

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

@Component({
    selector: 'person-input',
    template: `
      <input #personName type="text" />
      <button (click)="add(personName)">Add Person</button>
    `
})
export class PersonInput {
    @Output() addPerson = new EventEmitter();

    add(personInput){
      this.addPerson.emit(personInput.value);
      personInput.value = '';
    }
}
```

Container & presentational components



The image represents a component tree of a container component and its presentational components

See https://gist.github.com/btroncone/a6e4347326749f938510#utilizing-container-components for more information

Pipes

An Angular pipe is a function that transforms input values to output values for display in a view.

```
<!-- Today is Apr 21, 2017 -->
<span>Today is {{ currentDate | date }}</span>
```

- Pipes can be thought of as formatters
- Custom formatters can provide powerful functionality

The example represents formatting a date string or Date object as a string in the view

Services

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

@Injectable()
export default class DataService {
  public title: string = 'My Application';
}
```

Modules

Helps you organize an application into cohesive blocks of functionality. An Angular module identifies the components, directives, and pipes that the application uses along with the list of external Angular modules that the application needs, such as FormsModule.

Every Angular application has an application root-module class. By convention, the class is called AppModule and resides in a file named app.module.ts.

Modules use `NgModule` decorator

A feature module delivers a cohesive set of functionality focused on an application business domain, user workflow, facility (forms, http, routing), or collection of related utilities.

Module Example

```
import { NgModule, CommonModule } from '@angular/core';
import { HomeComponent } from './home.component';
import { HomeService } from './home.service';
export default class HomeModule {
  declarations: [
    HomeComponent
  ],
  imports: [CommonModule],
  exports: [HomeComponent],
  providers: [HomeService]
}
```

Introduction to RxJS

RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code.

- Observable
- Subscription

In computing, reactive programming is an asynchronous programming paradigm oriented around data streams and the propagation of change.

What does that mean?!

An event happens somewhere and an object responds to that event.

2 players in reactive programming: producer and consumer

Producer "emits" events to the stream Consumer has a subscription to the stream and reacts to new data

Good talk on Observables https://www.youtube.com/watch?v=DaCc8Ickuw8

Observables

The Observable object represents a push based collection.

Observables are used extensively in Angular (`Http` service)

Observables are used extensively in Angular. Methods on the `Http` service return `Observable<Response>`.

Subscription

A Subscription is an object that represents a disposable resource, usually the execution of an Observable.

Subscription objects have a single method!

RxJS Example

```
const source = Rx.Observable.timer(200, 100)
```

The example shows a timer that produces an initial value after 200ms. Subsequent events are produced every 100ms.

This example shows automatically unsubscribing

```
    .take(3);

const subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 200
// => Next: 100
// => Next: 100
// => Completed
```

Common RxJS Operators

Support for common higher order functions from JavaScript

- map
- filter
- reduce

Map - transform the items emitted by an Observable by applying a function to each item Filter - emit only those items from an Observable that pass a predicate test Reduce - apply a function to each item emitted by an Observable, sequentially, and emit the final value

Common RxJS Operators

- merge
- take
- forkJoin

merge - Combine any number of observables into a single observable take - Returns a specified number of contiguous elements from the start of an observable sequence forkJoin - Runs all observable sequences in parallel and collect their last elements.

Additional Resources

- André Staltz RxJS Introduction
  https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- Ben Lesh RxJS course
  https://egghead.io/instructors/ben-lesh

Ben Lesh also has a post about avoiding creating subscriptions:
https://medium.com/@benlesh/rxjs-dont-unsubscribe-6753ed4fda87

Angular HTTP Service

Http is available as an injectable class, with methods to perform http requests

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/map';
import { Hero } from './hero';
@Injectable()
export class HeroService {
  private heroesUrl = 'api/heroes';  // URL to web API
  constructor (private http: Http) {}
  getHeroes(): Observable<Hero[]> {
    return this.http.get(this.heroesUrl)
                    .map((res: Response) => res.json())
                    .catch(() => {
                      // Handle errors here
                    });
  }
}
```

Resources

- https://angular.io/docs/ts/latest/tutorial/
- https://angular.io/docs/ts/latest/guide/architecture.html