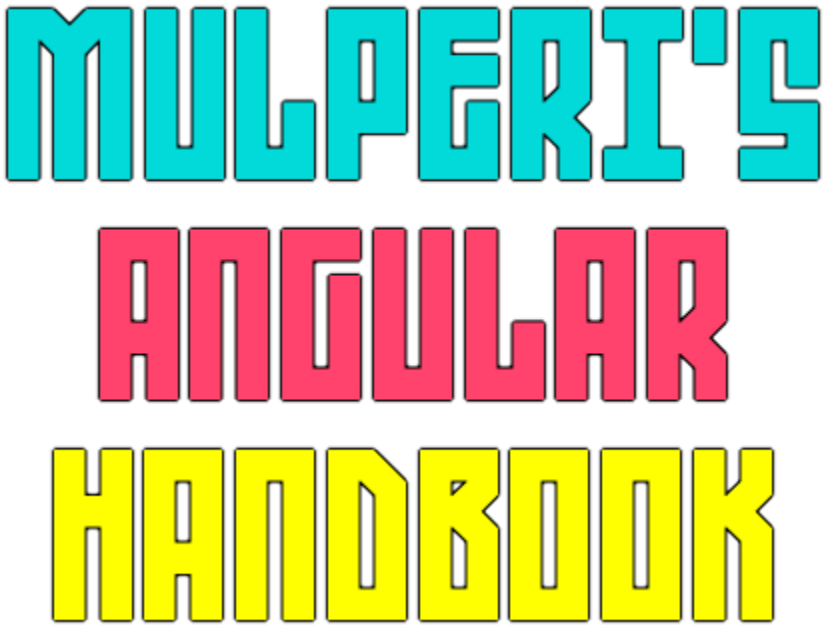
github.com

Mulperi/angular-handbook: Quickstart Angular with this handbook.

17–21 minutos



Mulperi's Angular Handbook

This handbook is a work in progress and meant to be used when training new Angular developers alongside with the official documentation, examples and exercices.

What is Angular?

Angular is a rapidly evolving JavaScript framework/platform for

modern web application development. It is open-source and led by Google's Angular Team. To be able to understand and do Angular development, you should know the basics of HTML, CSS and JavaScript.

The programming languages used in Angular development are TypeScript/JavaScript, HTML, CSS/SCSS.

Angular is not the same as AngularJS which is the first version of the framework and a whole different thing. When talking about just "Angular" we mean the latest version or Angular 2 and above.

TypeScript

TypeScript is the programming language used in Angular development. It's open-source and maintained by Microsoft. TypeScript is a superset of JavaScript which means that basic JavaScript code is also valid TypeScript. TypeScript is transcompiled to JavaScript.

> The biggest advantages of using TypeScript is the addition of static typing, even though it's optional and the ability to use advanced JavaScript features that are compiled down to a cross-browser friendly form. Basically you get cleaner code and catch error's early in development with TypeScript.

Angular vs others

Angular is often compared to React, Vue.js or other UI libraries.

One of the best arguments to go with Angular is that it is not just a library but a framework with all the tools you need to make your app from start to finish. It is "opinionated" in a way that you use all the tools Angular gives you like TypeScript language, Angular Router, HttpClient, RxJS with observables etc. You don't have to make these decisions yourself like with the lightweight libraries.

Single Page Application

Single page application or SPA is a web page/application that offers a desktop application-like user experience. The content is updated dynamically with JavaScript so there is no page reload when you switch to another page or sub page. This is what you make with Angular.

Building blocks of an Angular app

- Module
- Component
- Template (belongs to a component)
- Directives and pipes
- Service
- Routing
- Store (optional)

Decorator

A decorator and it's metadata tells Angular what to do with the class: is it a component, a module or a service etc.

Example of different kinds of decorators:

```
@NgModule({
    declarations: [AppComponent],
    imports: [BrowserModule]
    bootstrap: [AppComponent] // Only for root module
})
Export class AppModule {}

@Component({
    selector: 'app-list-item',
    templateUrl: './app-list-item.component.html',
    styleUrls: ['./app-list-item.component.scss']
})
Export class ListItemComponent {}

@Injectable({
    providedIn: 'root'
})
Export class MyDataService {}
```

Module

Angular applications are modular and always have at least one module: the root module. Modules can import data from other modules and declare components they are going to use. Modules

are classes with `@NgModule()` decorator and `.module.ts` file extension.

If you have a large application, you can split features into modules and load them "lazily" - in other words: only when an user selects the feature. This greatly reduces your application's initial loading time. It is good practice to only load modules that are needed or load them in the background with `PreloadAllModules` preloading strategy.

NgModule() decorator takes in an object with following properties (from https://angular.io/guide/architecture-modules):

- declarations: The components, directives, and pipes that belong to this NgModule.
- imports: Other modules whose exported classes are needed by component templates declared in this NgModule.
- exports: The subset of declarations that should be visible and usable in the component templates of other NgModules.
- providers: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
- bootstrap: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

Component and a view

Components are reusable custom elements you use in Angular application's html files. You define a component's selector (element name) in the `Component()` decorator metadata. Components have `.component.ts` file extension.

> You can change the component selector prefix inside angular.json settings file, it's app by default. You can also set the prefix when creating a new project: `ng new myapp --prefix myprefix`

Below is an example of some child components in app.component.html (the parent):

```
<app-header>
</app-header>
<app-item-list></app-item-list>
<app-footer></app-footer>
```

- A component defines part of the application logic in it’s class
- A template defines component’s view
- A template is regular html with Angular's own template-syntax
- A component is it’s `.ts` (class definition and component logic), `.html` (template/view) and `.scss` (styles) files

Example of a simple component.ts file with @Component() decorator:

```
@Component({
    selector: "app-item-list", // Will be used like
<app-item-list></app-item-list>
```

```
    templateUrl: "./item-list.component.html",
    styleUrls: ["./item-list.component.scss"]
})
export class ItemlistComponent {}
```

You can provide html file in the `templateUrl` property or you can alternatively provide inline html code to `template` property. It's usually better to have the template in a separate file though.

```
@Component({
selector: 'app-item-list',
template: `
    <div *ngFor="let item of items | async">
        {{ item.name }}
    </div>
`,
styleUrls: ['./item-list.component.scss']
})
export class ItemlistComponent {
    items: Observable<Item[]>;
}
```

Service

A service is a class with distinct purpose. A component delegates certain actions to a service. Like getting data from a server for example. If you are using a store in your application you usually want to call service from an **effect** and not from the component.

A service is a class with `@Injectable()` decorator and `.service.ts` file extension. In it's metadata you tell Angular where you want to provide the service. It is usually string `'root'`.

```
@Injectable({
    providedIn: 'root'
})
export class DataService {}
```

When Angular creates an instance of a component, it checks the component constructor function for what dependencies the component needs. This is dependency injection: the component is dependent on the service.

```
@Component({...})
export class AppComponent {
   constructor(private service: DataService) { }
}
```

Folder structure

There are different kinds of conventions on how to arrange your project directory. It's best to try to keep it organized and one way to help you do that is if you divide your components to **container components** and **presentational components**. This makes it easier to maintain code and find a specific application logic since most of the communication with the store or service, for example, is happening with the containers that pass down necessary data to the presentational components.

> *Container components are concerned with how things work. Presentational components are concerned with how things look.*
> Dan Abramov

An example of a project directory

```
src/
    app/
        components/
            footer/
            header/
            item-list/
        containers/
            main-page/
            settings-page/
        app.module.ts
```

Data binding

Data binding is a way to reflect data changes in one place to another place. There are few different kinds of data binding with each having their own syntax.

- Interpolation – Bind expression or component property in the template `{{data}}`
- Property binding – Bind expression to element attribute/child component property `[attributeName]`
- Event binding – Bind component to react to an user event like mouse click or custom event `(eventName)`

- Two-way binding – Component reacts to template event and also the template reflects what is happening in the component's TypeScript-file `[(data)]`

  Examples of data binding:

```
INTERPOLATION:      <li>{{ item.name }}</li>
PROPERTY BINDING:   <app-item-details
[item]="selectedItem"></app-item-details>
EVENT BINDING:      <li (click)="selectItem(item)">
</li>
```

Input() and Output()

If you want to pass down data to a child component, you need to use `@Input()` decorator in the child component's class. The input property name will be used as an attribute in the template side.

```
<!-- Passing string to welcomeMessage input -->
<my-component welcomeMessage="Hello">

<!-- Passing an expression/variable named Hello to
input -->
<my-component [welcomeMessage]="Hello">
```

If on the other hand you want to send data back to the parent component, the child component needs to emit the event and send data inside the $event object with EventEmitter using `@Output()` decorator.

```
export class ItemlistComponent {
```

```
    @Input()
    data: Item[];

    @Output()
    itemClick: EventEmitter<Item> = new
EventEmitter();

    onClick(item: Item) {
        this.itemClick.emit(item);
    }
}
```

When creating a custom event ("itemClick" in this case), you then need to listen and react to it in the parent component:

```
<app-item-list (itemClick)="onItemClick($event)">
</app-item-list>
```

The contents of an $event object depends on the event that is used. In this case it will be a "Item" that's passed as an argument to the `emit()` method. Then it's up to the parent component to deal with the event and it's data in it's `onItemClick()` method.

Directives and pipes

Directives modify the template dynamically. Their functionalities are defined in a class with `@Directive()` decorator. You can create your own directives or use Angular's default ones like *ngFor or *ngIf. Structural directives alter the DOM and are

marked with *.

```
<li *ngFor="let item of items">{{ item.name }}</li>
<app-item-details *ngIf="itemSelected"></app-item-details>
```

Pipes alter values in the template like date for example

```
<p>My birthday is {{ dateObject | date }}</p>
```

> Use `async` pipe to subscribe to an Observable from the template! This way you don't need to worry about unsubscribing and the component.ts file stays nice and clean.

Routing

Angular router enables you to navigate between views. Router takes the URL and navigates to corresponding component. Routing can be integrated to the root module in a simple application or it can be taken to it's own routing module which is usually preferred.

You can also route to a lazily loaded module that has it's own routing configurations.

Observable

Reactive programming is handling asynchronous data streams which can emit many values over time. You can create a stream from almost anything. RxJS - A JavaScript library for reactive programming using Observables is included in Angular. For

example, HTTP-requests return an Observable that can be subscribed to by calling the `subscribe()` method of the Observable and only then is the request actually made.

This concept is important to understand. Like Promises in JavaScript, Observables too are objects that you can pass around within your application and are just representations of an asynchronous operation. The request that is related to that specific object is executed when you call the subscribe method.

> *An Observable instance begins publishing values only when someone subscribes to it. You subscribe by calling the subscribe() method of the instance, passing an observer object to receive the notifications.* Angular.io

Example of an Observer object passed to a subscribe method to handle incoming data, errors or stream completion:

```
myObservable.subscribe({
    next: x => console.log('Observer got a next
value: ' + x),
    error: err => console.error('Observer got an
error: ' + err),
    complete: () => console.log('Observer got a
complete notification'),
});
```

To make this code shorter, you can pass in 1-3 of the functions in order like this:

```
myObservable.subscribe(
    x => console.log(x),
    err => console.error(err)
);
```

The selectors in NgRx also return observables from the state tree. This way when your components subscribe to state changes, you can immediately see data change in the state reflecting to the component and the view.

- Observable - A way to communicate between two parties: publisher and subscriber
- Operators - tools to modify the stream like combine or concatenate several data streams for example

Tools

Getting an Angular app running is very easy. Install the following tools to start development.

- Chrome & Redux DevTools extension
- Node.js
- Angular CLI

Node.js and npm

Angular uses npm - "world's largest software registry" - for dependency management. First install to get access to npm.

See file `package.json` for project dependencies and scripts.

Angular CLI

Angular has a good command line interface for managing your project. Install it globally via npm and then create and run new project.

```
npm install -g @angular/cli
ng new myapp --style scss --prefix myapp
cd myapp
ng serve
```

> If you are jumping into on-going project, you may not want to use ng serve to start the development server. See `package.json` for start script, it's usually `npm start`.

Chrome and Redux DevTools

For an application that utilizes NgRx/Store use Chrome extension called Redux DevTools to be able to view the state and what is happening in the store.

Exercises

Exercise 1

- Get to know the tools
- Create new Angular project
- Study the boilerplate (root folder, config files)
- Study a module and a component

- Generate a new component manually or with CLI and use the component
- Import and declare components in a module from a “barrell” file

Exercise 2

- Create new service
- Create mock data (JSON) in to assets folder and request it with HttpClient inside the service
- Use event binding in a component
- Use service inside a component
- Subscribe to an Observable inside a component or a template and compare the differences (remember to unsubscribe if you subscribe in component)
- Use structural directives *ngFor and *ngIf
- Use conditional styling in a template [class.active]="boolean”

Exercise 3

- Try communication between components using @Input and @Output
- Use property binding to send data to a child component
- Use EventEmitter to emit a custom event and react to it in the parent component
- Study Observable

Exercise 4

- Build a routing
- Test a route guard
- Create a module that is lazily loaded

i18n

> ***Internationalization*** *is the process of designing and preparing your app to be usable in different languages.* ***Localization*** *is the process of translating your internationalized app into specific languages for particular locales.* Angular.io

Angular has built in internalization tools that are very easy to use.

Example of an usage of i18n attribute in a template:

```
<h1 i18n="site header|An introduction header for this sample@@introductionHeader">Hello i18n!</h1>
```

```
i18n="context|information@@customid"
```

Exercise

- Study Angular i18n tools
- Use i18n syntax in a template

Extract the translation file, translate it and run development server with language configuration (you need to add the configuration to angular.json first too, see documentation for details):

```
ng xi18n --i18n-locale fr --output-path locale --out-
```

```
file messages.fr.xlf
```

Now, below all the `<source>` elements, add corresponding `<target>` elements and add the translated text inside.

```
ng serve --configuration=fr --aot
```

NgRx/Store

NgRx is a Redux inspired state management for Angular. In a larger application it is preferred to use something like ngrx to manage the state of the application.

Imagine the application state as a JavaScript object (key-value pairs) or a tree-like structure. Each key or branch in the state tree is a slice or feature of the state.

In addition to the state object, the store consists of:

- Actions
- Reducers
- Effects (optional but recommended)
- Selectors (optional but recommended)

Why store?

One of the main benefits of the store is that the application state is in one place and is abstracted. Any part of the state is accessible from anywhere in the application. This additional layer of abstraction of the state helps to keep the code clean and prevents too complex communication "chains" where components pass data

to child components.

For example, normally you would pass the data down to a child component with property binding like this:

```
<child-component [data]="data"></child-component>
```

With store, you dispatch an action in the parent component that updates the state with the data. In the child component you subscribe to the particular "slice" of the state that holds the data that component is interested in. Now the child component is always up to date with the latest data.

Updating application state

To update the state, you dispatch an `action`. The action goes into `reducer` function that return a new state if needed.

`Effects` also listen to actions and their job is to do asynchronous tasks and more complex logic on the background. Like getting data from the server via a service for example. Effects usually return new actions like "DATA_LOAD_SUCCESS". And those new actions again go to reducers and effects.

With `selectors` you define wich part of the state tree you want to subscsribe to.

```
ngrx/Store data flow (roughly)


Action > Reducer > Store > View
       > Effect > Action
```

Exercise

```
npm i @ngrx/store @ngrx/router-store @ngrx/effects
@ngrx/store-devtools ngrx-store-freeze
```

- Install Redux DevTools extension to Chrome
- Study the NgRx and how to apply the store into your Angular project
- Build a store and make sure that router-reducer can be seen in the state with DevTools
- Make a new branch (feature) to the state and actions, reducer, effects and selectors to it

Good practices

- Separating container and presentational components
- Block Element Modifier naming convention for CSS classes