Angular

Introduction

Angular is a Javascript framework for building client-side applications (HTML/CSS/Javascript). Your applications can be targeted to any device that supports a web stack: Web browser, Phone Gap, Cordova, etc..

Why Angular over other frameworks? Angular provides support for...

- HTML data binding.
- control structures in HTML (e.g. loops and ifs).
- older HTML engines / older web browsers.
- communicating with a backend service (e.g. HTTP post).
- modular software design.

> **NOTE**: Although Angular is a Javascript framework, the prefered language for Angular is TypeScript. Remember that TypeScript compile down to Javascript.

Setup

Much like everything else in the Javascript ecosystem, you need to use NodeJS+npm to get up and running with Angular.

> **NOTE**: If you need a refresher on NodeJS+npm or TypeScript, see the relevant documents.

Install

To do anything with Angular, you need to use the Angular CLI. You can install the Angular CLI globally by using NodeJS and npm: npm install -g @angula/cli. Once you have it, you can create a new Angular project by doing ng new <appname>.

See the Angular CLI section for more information.

IDEs

For IDEs, you can use any but VSCode seems to be the big one. VSCode has builtin support for TypeScript (the default language of Angular), but make sure to install the tslint extension and the typescript hero extension.

tslint will give you immediate linter feedback and typescript hero lets you easily do things like auto-import and organize imports.

All the necessary linting/editor configurations will be generated for you by the Angular CLI when you use it to create a new project. VSCode will recognize and use them automatically.

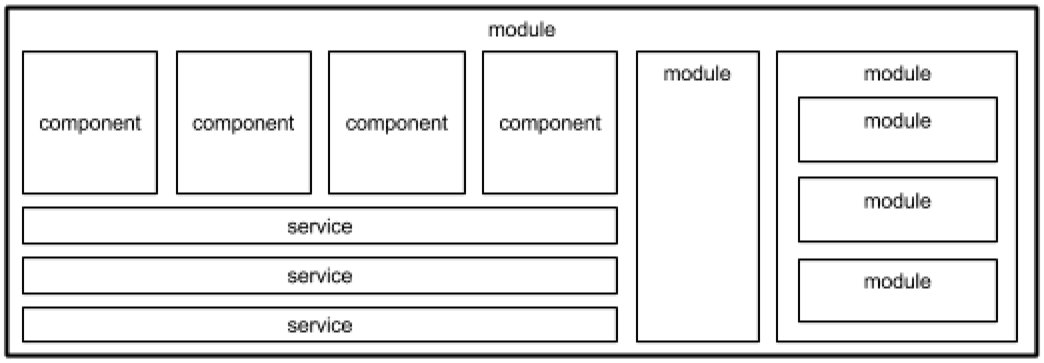
Running Code

You can use the Angular CLI to spin up and run your code: ng serve -o. You can also use the start script in your package.json file (which delegates to the same command).

See the Run Application subsection of the Angular CLI section for more information

Architecture

The overall architecture of any Angular application is as follows. Applications are broken down into modules, components, and services.



Modules can import components and services from other modules, as well as export their components and services to other modules. Components talk with each other and the outside world via services.

An Angular module/component/service is implemented as TypeScript class with decorators.

Modules

Modules in Angular aren't like modules in TypeScript or ES2015. That is, they're more about organizing your application into conceptual blocks vs importing/exporting code from individual files.

In Angular, you can place multiple components into a module. You can also stick multiple modules together into a shared module, such that if you're frequently pulling in the same set of modules you can change it to just pull in a single module.

A module is defined as an empty class with an NgModule decorator that sits in a file with a .module.ts extension. The following is an example of a barebones Angular module. It has 1 component in it called AppComponent, and it's intended to run on the browser …

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

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

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

What do all the fields in NgModule decorator mean?

Bootstrap → There can be multiple Angular modules in your application, but every application has a root module (the top-level module). That root module has a root component, and that root component is what's passed into the bootstrap field. This component is what initially gets loaded and rendered out to the browser -- it's basically the starting point of your application.

Declarations → Every component we create belongs to one (and only one) module. The module knows what it owns via the declarations field. This is how it becomes visible to other components / pieces of the module.

Exports → Components in the module can be shared with other modules via the exports field. When a module exports components, those components become visible to any other module that imports that module.

Imports → Other modules can be imported by a module by adding them to the imports field. When a module is imported, all of its exports become visible to the components/pieces in the module and it can do additional work behind-the-scenes to set things up or provide some extra functionality.

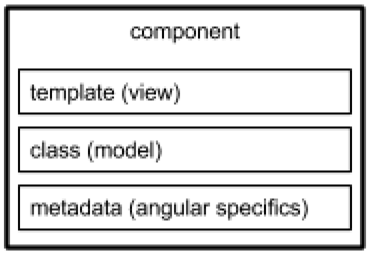
Providers → Services are declared in the providers field. Unlike the declarations field, services in the provider field are registered globally as a singleton. Any module can inject your service, even if your module hasn't been imported by that module.

> **NOTE**: Why is bootstrap an array?
>
> **NOTE**: IMPORTS ARE NOT INHERITED -- Importing modules does NOT provides access to the imports of those modules. If you want to do that, you must explicitly re-export what you've imported. This may be useful if you're consolidating features from many modules.
>
> **NOTE**: Pipes and directives also belong to modules and are specified in the declarations field. See the pipes section to learn more. They're not talked about here to keep things simple.

In our example… AppComponent gets bootstrapped because it's the root component, and BrowserModule gets imported (this is a Angular module provided by Angular) because our module+components are intended for a browser and BrowserModule does some behind-the-scenes work to make the browser work.

Components

Components have a template frontend that handles the view and an operational backend that handles the logic. The frontend portion is an HTML template enhanced using Angular's markup, while the backend portion is a Javascript class. There may also be some angular-specific metadata associated with the component.



A component is defined as a class with an Component decorator that sits in a file with a .component.ts extension. In addition, a component may come accompanied with a unit testing file that has a .component.spec.ts extension.

The following is an example of a barebones Angular component. The template for the component is embedded in the annotation...

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

@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1>{{msg}}</h1>
     <div>...other stuff here...</div>
   </div>
 `
})
export class AppComponent {
 msg: string = 'Test Page';
}
```

Note that the example component decorator above...

- embeds the HTML template to use
- provides fields being referenced by the HTML template (msg in this case)
- declares a selector (also called directive) to identify where to render the template

Ultimately, your component templates will end up either being injected into other higher-level components or getting dumped into the main HTML file…

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8">
 <base href="/">
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
 <app-root></app-root>
</body>
</html>
```

Notice how in the above HTML block, we have a app-root tag. In the component example above, our selector was set to app-root. As such, when Angular actually runs, the app-root tag in the HTML block above will automatically get injected with our component's template.

Templates

Templates refer to the view of a component -- the HTML portion of a component. A template uses special Angular tags to display data from the component's model, as well as provides functionality for basic control structures (e.g. looping, conditionals, etc..).

HTML

There are 2 ways to define the HTML template for your component. The first way is to embed the template directly into the Component decorator's via the the template property…

```
@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1>{{msg}}</h1>
     <div>...other stuff here...</div>
   </div>
 `
})
```

The template tag is good for small templates, but larger template should extract the template data into a separate file and use the templateUrl property…

```
@Component({
   selector: 'app-products',
   templateUrl: './product-list.component.html'
})
```

The problem with embedding is that it's errorprone for larger chunks of code because you don't get IDE features such as Intellisense and being notified of malformed blocks.

Styles

> **NOTE**: You don't have to use this. You can always just include the styles directly into your template or link to a stylesheet from your final HTML. This seems to be the preferred way for Angular though.

There are 2 ways to define the CSS that for the HTML template of your component. The first way is to embed the styles directly into the Component decorator's via the the styles tag…

```
@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1>{{msg | uppercase}}</h1>
     <app-products></app-products>
   </div>
 `,
 styles: ['h1 {color: red}]']
})
```

The styles tag is good for small templates, but larger template should extract the CSS data into a one or more CSS files and use the styleUrls property (array)…

> **NOTE**: Remember that this is styleUrls, not stylesUrl.

```
@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1>{{msg | uppercase}}</h1>
     <app-products></app-products>
   </div>
 `,
 styleUrls: ['produst-list.component.css']
})
```

Binding

Angular provides several ways to bind data from your model (component class) to your view (component template).

Interpolation (Model → View)

You can bind portions of your view (component template) to your model (component class) by using interpolation binding. Interpolation bindings expressions encapsulated by double curly braces within the HTML template: e.g. {{ expression }}. These are called template expressions.

These expressions can pull fields, invoke functions, and even execute simple expressions. In the following example, the expression {{'Page name:' + msg}} pulls in the data from the msg field and prepends 'Page name:' to it.…

```
@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1>{{'Page name:' + msg}}</h1>
     <app-products></app-products>
   </div>
 `
})
export class AppComponent {
 msg: string = 'Test Page';
}
```

Interpolation expressions are intended to be read-only. That means that you can write them out to the view, but user interactions with the view won't necessarily update the model.

Property (Model → View)

Property binding is like interpolation binding, except that it's defined differently. Instead of using double curly brackets ({{ }}), you wrap the property of the HTML element you want to inject into with square brackets and set the value of that property to a template expression (see Interpolation section above for more info on template expressions).

> **NOTE**: It looks like property means DOM property, which can be an attribute or some other DOM class. For example, the src attribute on an img tag may be a property, or style.width.px may be a property. More info on this topic can be found here: https://dzone.com/articles/property-bindings-in-angular-2, but the jist of it is that class and style attributes aren't treated as text by the DOM. They are a tree structure which you can drill down into.
>
> **NOTE**: A property may also be the property of a nested component (a component which you've added to your component). See the section on nested components for more info.

Property bindings seem to be preferred over interpolation binding, however in certain cases you can only use interpolation binding... online comments are saying that you should use this method over Interpolation binding if your expression doesn't evaluate to a string. However, I've tried feeding in a number into Interpolation binding and it worked fine?

The following example sets the innerText attribute of h1 to the exact same expression as the example in the Interpolation section…

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

@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1 [innerText]="'Page name:' + msg"></h1>
     <app-products></app-products>
   </div>
 `
})
export class AppComponent {
 msg: string = 'Page Title';
 func(input: string) { return input + 'test'; }
}
```

> **NOTE**: Instead of square brackets ([ ]), you can do the same thing by just prepending the "bind-" to the attribute. So in the above example, instead of [innerText] it would be bind-innerText. I haven't had a chance to test this.

Events (Model ⟵ View)

Event bindings are the opposite of interpolation/property bindings. Instead of writing out your model to your view, they relay events on your view to your model. For example, a user clicking a button may trigger a method call on your model.

To bind events, in the tag for which you're intercepting the event, wrap the event in normal parenthesis and set the value to a template expression (see Interpolation section above for more info on template expressions).

> **NOTE**: Your template expression will almost always be a method call on your model.
>
> **NOTE**: These events aren't unique to Angular. They're either DOM events or events that are custom to the container. For a full list of DOM events, see https://developer.mozilla.org/en-US/docs/Web/Events. To learn more about container events, see the Nested Components section.

Here's an example that calls a method when a button is pressed. The method sets a field property, which should instantly update the DOM (the browser will show the new field)...

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

@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1>{{msg}}</h1>
     <button (click)="func('wtf')">Change msg to wtf</button>
     <app-products></app-products>
   </div>
 `
})
export class AppComponent {
 msg: string = 'Page Title';
 func(input: string): void { this.msg = input; }
}
```

Two-way (Model ↔ View)

Two-way binding is an amalgamation of property binding and event binding. It binds an input tag's value such that...

- if the user edits the input, your model will update.
- if you update your model, the input will update.

For two-way binding, in the input tag for which you're binding, add [(ngModel)]=templateexpression as an attribute on that tag (see Interpolation section above for more info on template expressions).

> **NOTE**: ngModule is a directive inside Angular's FormsModule (cause it's most often used with form data). If you want to use it, FormsModule needs to be imported by the Module which your component belongs to….
>
> ```
> @NgModule({
>  declarations: [ ... ],
>  imports: [ BrowserModule, FormsModule ]
>  providers: [ … ],
>  bootstrap: [ … ]
> })
> ```

Here's an example of two-way binding. When the msg field gets updated by the text input and vice versa...

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

@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1>{{msg}}</h1>
     <input type="text" [(ngModel)]="msg"/>
     <button (click)="func('wtf')">Change msg to wtf</button>
     <app-products></app-products>
   </div>
  `
})
export class AppComponent {
 msg: string = 'Page Title';
 func(input: string): void { this.msg = input; }
}
```

Pipes

Often when you're binding data, you want to transform it somehow before being displayed. For example, if you're dealing with currency you may want to format your number such that it has a dollar sign ($) in front of it and it truncates the fractional digits to 2.

This is what pipes do. To use pipe, use the pipe symbol (|) in your template expression. This works just like pipes in linux (e.g. cat somefile | grep somestring). You pipe the output of one expression to another.

Using

Common pipes that come with angular include…

- uppercase
- lowercase
- date
- json
- slice
- currency

Here's an example that uses pipes to convert to uppercase…

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

@Component({
 selector: 'app-root',
 template: `
```

```
    <div>
      <h1>{{msg | uppercase}}</h1>
      <app-products></app-products>
    </div>
  `
})
export class AppComponent {
 msg: string = 'Page Title';
}
```

Some pipes support parameters. Pipe parameters are defined using colons. For example {{ price | currency:'CAD' }} will mark the number as Canadian currency.

Creating

To build custom pipes, you create a class that implements the PipeTransform interface and uses the Pipe decorator…

```
@Pipe({
   name: 'test'
})
export class TestPipe implements PipeTransform {
   public transform(value: string, args: string): string {
       return value.replace(args, '_');
   }
}
```

The first parameter is the input and the second parameter is the arguments to the pipe (could be multiple? This can be an array?).

Standard Angular conventions apply to pipes. That means you'll need to…

- declare your pipe in your module (just like with your components/services/etc..).
- put your pipe in a file that has a .pipe.ts extension.
- suffix the class name with Pipe.

Control Structures

A structural directive manipulates the structure/layout of the view by adding, removing, or shifting around elements and their children. They're essentially programming control structures, but embedded directly within the view/template.

> **NOTE**: The selectors for the components we created are also referred to as directives. These structural directives are directives supplied by Angular's BrowserModule. You need to import the BrowserModule into your own module if you want access to them…

```
@NgModule({
 declarations: [ ... ],
 imports: [ BrowserModule ]
 providers: [ … ],
 bootstrap: [ … ]
})
```

Structural directives always start with an asterisk (*).

If (*ngIf)

*ngIf is a structural directive that only shows the tag (and its children) it’s been applied to if the expression supplied to it evaluates to true. For example…

```
<h1 *ngIf="msg == 'hi'">{{msg}}</h1>
```

The above tag will only get added to the DOM if the component class has a field named msg and that field is equal to the string 'hi'.

> **NOTE**: Remember you need BrowserModule to use this.

For (*ngFor)

*ngFor is a structural directive that repeats whatever tag (and its children) it’s been applied to once for each item in an array / iterable list. For example…

```
<h1 *ngFor="let message of messages">{{message}}</h1>
```

The above tag will get output into the DOM once for each item in the field named messages.

> **NOTE**: Remember you need BrowserModule to use this.
>
> **NOTE**: In the above example, we’re using of instead of in. Apparently of was chosen to match Javascript syntax. Replace of with in won’t work. Javascript arrays are considered objects and their indexes are technically properties. If you use in, you’re iterating over the properties of an object rather than the elements of an array / iterable list. To iterate over the elements, you need to use of instead of in.
>
> Still having trouble thinking about this?
>
> The example below will print out 0 1 2 to the console…
> let x = ['a', 'b', 'c'];
> for (val in x) {

```
    console.log(val);
}
```

The example below will print out a b c to the console…

```
let x = ['a', 'b', 'c'];
for (val of x) {
    console.log(val);
}
```

Lifecycle

You can make it so Angular notifies your component when it enters different stages of its lifecycle. For example, you can make it so that a component gets notified once it's been initialized.

To get notified of a lifecycle event, your component class needs to implement the interface for that event. For example, if you want your component to be notified on initialization, you would implement the OnInit interface…

```
@Component({
 selector: 'app-root',
 template: `
   <div>
     <h1>{{msg | uppercase}}</h1>
     <app-products></app-products>
   </div>
 `
})
export class AppComponent implements OnInit {
   msg: string = 'Page Title';

   public ngOnInit(): void {
       console.log('initialized!!!');
   }
}
```

Typical lifecycle events include…

- onInit → component has been initialized (do prepwork for component here)
- onChanges → on change to component input property (see nested components section)
- onDestroy → component has been destroyed (do cleanup for component here)

Nesting (Communication)

In certain cases, you may be pulling in other components into your component (nesting components). The standard way to communicate between components is to use services, but if you're nesting components you can also communicate via the properties of the components you're nesting. This is done via event binding and property binding.

> **NOTE**: See the section on bindings if you need a refresher on event and property bindings.

If you want to expose a property in your component as a …

- property, use the @Input decorator on that property
- event, use the @Output decorator on that property and make sure its an EventEmitter

> **NOTE**: The @Input/@Output decorators need to be invoked -- that is, when you apply them on your property, you must call them as if they're functions. This'll become more clear in the example below.

In addition to that, your component should implement the OnChanges lifecycle interface to get notified when parent component updates a property in the child. You can do any extra work required when a property gets changed by the parent container via OnChanges (e.g. further computations or changes to the component).

> **NOTE**: Forgot about EventEmitters? Refer to the NodeJS doc. This EventEmitter class is specific to Angular though -- it's imported from @angular/core.

> **NOTE**: Forgot about lifecycle interfaces? See the Component LifeCycle section.

Here's an example of a component that's nested in another component…

```
@Component({
   selector: 'app-nested',
   template: `
   <div (click)='notifyOfClick()'>{{message}}</div>
   `
})
export class NestedComponent implements OnChanges {
   @Input()
   message: string = 'initialmsg';

   @Output()
   messageClick: EventEmitter<void> = new EventEmitter<void>();
```

```
    notifyOfClick() {
        this.messageClick.emit();
    }

    ngOnChanges(changes: SimpleChanges): void {
        // don't need to do anything here
    }
}
```

```
@Component({
 selector: 'app-root',
 template: `
   <div>
     <app-nested (messageClick)="alert()" [message]="msg"></app-nested>
   </div>
 `
})
export class AppComponent {
 msg: string = 'My Custom Message!';
 alert() {
   console.log('nested component clicked');
 }
}
```

Note what's happening in the nested container…

- message property is exposed using the @Input() decorator.
- messageClick event is exposed using the Output() decorator.
- template outputs message property to a div
- template triggers notifyOfClick() when that div is clicked, which invokes messageClick.

Note what's happening in the parent container. The child container (pm-nested) was added into the template along with…

- an event binding that listens for messageClick events on the nested container.
- a property binding that binds to the message property of the nested container.

NOTE: Remember that event bindings are wrapped in parenthesis while property bindings are wrapped in square brackets.

Routing

> **NOTE**: This should be a document of its own. There's a lot going on here and some gotchas in terms of lazy loading and service injection and other things. More research is needed.

Component routing allows a component to be swapped out for another based on the URL. This allows the user to manually navigate to parts of your application via the URL they use (e.g. bookmarks). You can also internally reference these paths for things like navigation buttons.

To use of routing, you need to import the RouterModule into your module AND configure it for the routes of your application. The following example is from the pluralsight Angular lesson...

```
@NgModule({
 declarations: [... ],
 imports: [
   RouterModule.forRoot([
      { path: 'product', component: ProductListComponent },
      { path: 'product/:id', component: ProductDetailComponent },
      { path: 'welcome', component: WelcomeComponent },
      { path: '', redirectTo: 'welcome', pathMatch: 'full' },
      { path: '**', redirectTo: 'welcome', pathMatch: 'full' }
    ], { useHash: true })
  ]
 ],
 providers: [...],
 bootstrap: [...]
})
export class AppModule { }
```

In addition, in your index.html you need a base element with a path specified…

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8">
 <title>A</title>
 <base href="/">

 <meta name="viewport" content="width=device-width, initial-scale=1">
 <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
 <app-root></app-root>
```

```
</body>
</html>
```

Note that in the router configuration...

- each path is being mapped to a component or redirected to another path
- paths can take in arguments -- e.g. product/:id -- id is the parameter
- paths can be glob patterns -- first match in the list is the one used
- useHash is set to true -- this is so the URL rewriting uses anchors rather than rewriting the URL in such a way that your web server needs to be configured

When you add the RouterModule, you get access to new directive called routerLink and router-outlet.

routerLink allows you navigate to routes in your template...

```
<a [routerLink]="['/welcome']">Home</a>
```

> **NOTE**: The value for a routerLink directive is a string in an array. That's why the square brackets and quotes are there. Parameters can be added to this array (e.g. the /product/:id path in the RouterModule example above)

router-outlet is where the current route is rendered in your template…

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

Services

All code that doesn't belong to a component should go into a service. How do we know what code belongs in a component? A component's model should focus mainly on managing that component. It shouldn't be interfacing directly with the backend, other components, or third-parties.

Services are an abstracted way to interface/bridge to the backend, other components, and third-parties. For example, hitting a data source to get/set the data used by component would be done through a service. The benefit of this is that it hides the implementation details, and the service can be mocked out for unit tests.

A service is defined as a class…

- with an Injectable decorator.
- in a file that has a .service.ts extension.

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

```
@Injectable()
export class MessageService {
  messages: string[] = [];

  add(message: string) {
    this.messages.push(message);
  }

  clear() {
    this.messages = [];
  }
}
```

Note that the @Injectable decorator tells Angular that this service might itself have injected dependencies. According to the Angular documentation, even if it doesn't, it's good practice to keep it in place.

If you want to use this service, you need to add is a provider on a component or module. Once that's done, any component under that component/module can take in the service via the component class's constructor.

To add this service, add it as an element to the providers field of either the Component decorator or the Module decorator.

Remember that if you add it to the …

- Module decorator's provider field, the service will be available globally (even to other modules that don't import that module).
- Component's decorator's provider field, any nested component class (down the tree) can pull in the service.

```
@Component({
 selector: 'pm-root',
 template: `
   <div>
     <h1>{{msg}}</h1>
     <div>...other stuff here...</div>
   </div>
 `,
 providers: [ MessageService ]
```

> **NOTE**: It looks like the service instance is a singleton -- any component, regardless of it's the one that declared it as a provider or a child of that container, will be getting the same instance.

The service can then be pulled into a component via that component's constructor...

```
constructor(private MessageService messageService) {
    ...
}
```

> **NOTE**: Even though you're assigning the service in the constructor, you should not access the service until OnInit happens -- see the section on Component Lifecycle for more information.

Common Modules

CommonModule / BrowserModule

CommonModule gives access to the structural directives (*ngIf and *ngFor) and a bunch of other things. You may remember from the components section that we imported the BrowserModule to get access to these. There seems to be some confusion about what to use where.

https://angular.io/guide/ngmodule-faq#should-i-import-browsermodule-or-commonmodule seems to say that we should use BrowserModule in our root module, and CommonModule in any feature modules? Unsure about this.

To use the stuff provided by CommonModule, you need to import it into your module...

```
@NgModule({
 declarations: [...],
 imports: [
   CommonModule
 ],
 providers: [...],
 bootstrap: [...]
})
export class AppModule { }
```

Here are examples of the structural directives provided by CommonModule...

```
<div *ngIf="msg == 'hi'">{{msg}}</div>
<div *ngFor="let message of messages">{{message}}</div>
```

FormsModule

> **NOTE**: This should be a document of its own. There's a lot going on here and some gotchas in terms of lazy loading and service injection and other things. More research is needed.

FormsModule gives access to form-related functionality (e.g. validation?), but the most useful thing in this module is the two-way binding directive. See the two-way binding section for more information.

HttpClientModule

One of the most used services that comes with Angular is the HTTP service. The HTTP service lets you make HTTP calls to a server.

To use the HTTP service, you need to import the HttpClientModule into your module...

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

The HttpClientModule itself is what registers the service for injection. By pulling it into your module, any component/service under your module will be able to inject HttpClient via the normal constructor dependency injection method of Angular…

```
constructor(private _http: HttpClient) {
}
```

HttpClient provides all the common HTTP methods…

- GET
- POST
- DELETE
- PUT
- HEAD

It parses JSON as the response and it's does HTTP calls asynchronously. The return value of any of the above methods is an Observable object (from the RxJS project) which can be used to pipe results to observers as it comes in…

```
this._http.get<MyObject[]>('http://someurl/someservice')
       .do(data => console.log('All: ' + JSON.stringify(data)))
       .catch(err => {
           console.log(err.message);
           return Observable.throw(err);
       }.subscribe(
           response => this.values = response,
           error => this.errorMessage = <any>error
       );
```

Angular CLI

The Angular CLI is the preferred way of dealing with Angular. It's a one-stop shop for doing everything you need to do with Angular.

To install Angular CLI, you can use npm. To install globally, use sudo npm install -g @angular/cli. Once installed, you can execute the CLI by running ng.

> **NOTE**: If you don't want to install globally, take out the -g flag and npm will install locally. You can still access the ng script, but it'll be in ./node_modules/.bin.

You can access it via the ng command...

```
$ ./node_modules/.bin/ng
ng build <options...>
  Builds your app and places it into the output path (dist/ by default).
  aliases: b
  --target (String) (Default: development) Defines the build target.
    aliases: -t <value>, -dev (--target=development), -prod
(--target=production), --target <value>
  --environment (String) Defines the build environment.
    aliases: -e <value>, --environment <value>
  --output-path (Path) Path where output will be placed.
    aliases: -op <value>, --outputPath <value>
...
```

> **NOTE**: Remember that all the npm scripts shown in the setup section just delegated to to the Angular CLI (ng).

Running ng by itself will generate help text. The help text is excessively long. To get help on a specific angular command, use ng <command> --help.

The following subsections detail common Angular CLI tasks.

Create Application

To create a new module use ng new <appname>.

So for example, if you wanted to create a new app called myApp…

```
$ ng new myApp
  create myApp/README.md (1021 bytes)
  create myApp/.angular-cli.json (1241 bytes)
  create myApp/.editorconfig (245 bytes)
  create myApp/.gitignore (516 bytes)
  create myApp/src/assets/.gitkeep (0 bytes)
  create myApp/src/environments/environment.prod.ts (51 bytes)
  create myApp/src/environments/environment.ts (387 bytes)
  create myApp/src/favicon.ico (5430 bytes)
  create myApp/src/index.html (292 bytes)
  create myApp/src/main.ts (370 bytes)
  create myApp/src/polyfills.ts (2405 bytes)
  create myApp/src/styles.css (80 bytes)
  create myApp/src/test.ts (1085 bytes)
  create myApp/src/tsconfig.app.json (211 bytes)
  create myApp/src/tsconfig.spec.json (304 bytes)
  create myApp/src/typings.d.ts (104 bytes)
  create myApp/e2e/app.e2e-spec.ts (288 bytes)
  create myApp/e2e/app.po.ts (208 bytes)
  create myApp/e2e/tsconfig.e2e.json (235 bytes)
  create myApp/karma.conf.js (923 bytes)
  create myApp/package.json (1311 bytes)
  create myApp/protractor.conf.js (722 bytes)
  create myApp/tsconfig.json (363 bytes)
  create myApp/tslint.json (3040 bytes)
  create myApp/src/app/app.module.ts (316 bytes)
  create myApp/src/app/app.component.css (0 bytes)
  create myApp/src/app/app.component.html (1141 bytes)
  create myApp/src/app/app.component.spec.ts (986 bytes)
  create myApp/src/app/app.component.ts (207 bytes)
Installing packages for tooling via npm.
added 1149 packages in 21.482s

Installed packages for tooling via npm.
Successfully initialized git.
```

```
Project 'myApp' successfully created.
```

Note that this is a complete development setup, including…

- TypeScript compiler configuration (tsconfig.json)
- TypeScript linting rules (tslint.json)
- Karma unit testing configuration (karma.conf.js)
- Proactor end-to-end testing configuration (protractor.conf.js)
- editor configuration (.editorconfig)
- npm package.json along with all the required dependencies
- a base index.html (will almost never need to be touched)
- a base module and base component (app.module and app.component)
- all required polyfills
- setting up a git repository

This is taking 90% of the headaches out of developing a JS application. All you need to do to get working is start VSCode (install the tslint and typescript hero extensions if not already done so) and open up the folder. It's already decided everything for you in terms of how you should be coding / how you should be testing / what your editor should do about your spacing and braces / etc..

Here are some important things you should know when you're creating a new project…

- Style type can be specified via --styles option

  For example, you can choose to use SASS as your style engine. Angular will automatically set up webpack to compile your SCSS files via the SASS engine and bundle them.

  You can also change style after the fact by editing the apps.defaults.styleExt section of the .angular-cli.json file.

- Selector prefixes for your component can be set via the --prefix option.

  The Angular linter checks to make sure all your component selectors have a prefix. By default this prefix is app, but you can set it to whatever you want using this tag.

  You can also change the prefix after the fact by editing the apps.prefix section of the .angular-cli.json file.

- Setup routing for your project via the --routing option.

  Remember that routing is complex to use and has a few gotchas. By using this option, the CLI will automatically try to set up routing in your application.

- Add dirs to include into your app via apps.assets in the .angular-cli.json file.

- Add css files to include into your app via apps.styles in the .angular-cli.json file.

  NOTE: Other flags you may find useful: --skip-git --skip-tests --dry-run

Also, you should also update the following scripts in the generated package.json file...

```
"scripts": {
  "start": "ng serve -o",
  "lint": "ng lint --type-check"
}
```

The start script gets a -o flag added to make ng start a browser and the lint script adds a flag to do stricter checks.

Create Module

To create a new module use ng generate m <module_path>.

So for example, if you wanted your new component to be called apple.module.ts and sit in the test folder…

```
$ ng generate m test/apple
installing module
  create src/app/test/apple/apple.module.ts
  WARNING Module is generated but not provided, it must be provided to be
used
```

Here are some important things you should know when you're creating a new component…

- If you don't want a folder being created, add --flat.

  For example, if you wanted to create the example above directly in test/ instead of test/apple/, add the --flat flag.

- Automatically add the module as an import to another module via -m <module_name>.

  That's what the warning on the last line of the example is complaining about.

- Add unit testing stubs for the module using --spec true.

Remember that unit testing stubs are automatically created for services,components,etc.., but not for modules.

- Setup routing for your module via the --routing option.

  This will end up creating 2 new modules, a normal module and a routing module that gets imported to the routed module and sets up routing. If you use this, make sure you created your project with the --routing option so everything gets wired up correctly.

Create Component

To create a new component use ng generate c <component_path>.

So for example, if you wanted your new component to be called funny.component.ts and sit in the test folder…

```
$ ng generate c test/funny.component
installing component
  create src/app/test/funny/funny.component.css
  create src/app/test/funny/funny.component.html
  create src/app/test/funny/funny.component.spec.ts
  create src/app/test/funny/funny.component.ts
  update src/app/app.module.ts
```

Here are some important things you should know when you're creating a new component…

- If you don't want a folder being created, add --flat.

  For example, if you wanted to create the example above directly in test/ instead of test/funny/, add the --flat flag.

- If you want the template HTML/style to be embedded, add --inline-template.

- If you don't want unit testing stubs generated, add --spec false.

Note how the Angular CLI creates all the files and folders for the component as well as updates the module for which this new component sits in. The module will have this new component added to its declarations property…

```
@NgModule({
 declarations: [
   FunnyComponent
 ],
 imports: [...],
 providers: [],
 bootstrap: [...]
})
export class AppModule { }
```

Create Service

To create a new component use ng generate s <service_path> -m <module_path>.

> **NOTE**: Unlike with components, creating a service with ng doesn't create a folder for the service by default. This means that you don't need to add a --flat option.
>
> **NOTE**: Unlike with components, services are NOT automatically registered with the module. That's what the -m flag is for. There is no flag to register the service created with a component. If you want to do that, you need to manually add it to the Component decorator's provider field.

So for example, if you wanted your new service to be called funny.service.ts and sit in the test folder…

```
$ ng generate s funny.service -m app.module
installing service
  identical src/app/funny.service.spec.ts
  identical src/app/funny.service.ts
  update src/app/app.module.ts
```

Note how the Angular CLI creates all the files and folders for the service as well as updates the module for which this new component sits in (that's what the -m flag was for). The module will have this new component added to its declarations property…

```
@NgModule({
 declarations: [...],
 imports: [...],
 providers: [ FunnyService ],
 bootstrap: [...]
})
export class AppModule { }
```

Create Other

The CLI allows you to create other stuff as well (e.g. custom pipes). This is all done via the ng generate command. Use ng generate --help to get more information.

Run Application

To run your application, use the ng serve -o command. This compiles a development build of your application, spawns a development server to run it on, and starts up a browser. It also watches and rebuilds each time a source file is changed + refreshes the browser it opened...

```
$ ng serve -o
** NG Live Development Server is listening on localhost:4200, open your
browser on http://localhost:4200/ **
 12% building modules 24/27 modules 3 active
...myApp/node_modules/webpack/hot/log.jswebpack: wait until bundle
finished: /
Date: 2017-12-14T00:58:35.520Z
Hash: 8196e104a6a989ac9562
Time: 10035ms
chunk {inline} inline.bundle.js (inline) 5.79 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 20.2 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 547 kB [initial]
[rendered]
chunk {styles} styles.bundle.js (styles) 33.7 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 7.14 MB [initial] [rendered]

webpack: Compiled successfully.
```

NOTE: If you want to run your application as if it was in production (with AOT compilation and all the inlining and optimizations and stuff), use the --prod flag here. Remember that when you use this flag, you're also going to be pulling in production environment settings specified in the src/environments/environment.prod.ts file vs the normal development environment settings in the src/environments/environment.ts file.

NOTE: What gets built is in-memory. No output files are generated when you run your build like this.

Run Unit Tests

To run unit tests, use the ng test command. This command compiles the application and launches a browser to run the unit tests. This command also watches for changes to your source file -- if it detects a change it'll recompile and immediately re-run the tests...

```
$ ng test
 10% building modules 1/1 modules 0 active13 12 2017 17:17:33.847:WARN
[karma]: No captured browser, open http://localhost:9876/
```

```
13 12 2017 17:17:33.858:INFO [karma]: Karma v1.7.1 server started at
http://0.0.0.0:9876/
13 12 2017 17:17:33.860:INFO [launcher]: Launching browser Chrome with
unlimited concurrency
13 12 2017 17:17:33.867:INFO [launcher]: Starting browser Chrome
13 12 2017 17:17:33.868:ERROR [launcher]: No binary for Chrome browser on
your platform.
  Please, set "CHROME_BIN" env variable.
13 12 2017 17:17:37.623:WARN [karma]: No captured browser, open
http://localhost:9876/
13 12 2017 17:18:23.176:INFO [Firefox 57.0.0 (Ubuntu 0.0.0)]: Connected on
socket ufb7Dws8Er_k85clAAAA with id manual-7598
```

NOTE: This seems to want to run on Chrome by default. If it can't find Chrome, it'll prompt you to open up a browser and navigate to a URL.

Here are some important things you should know when you're running unit tests…

- If you only want the tests running one-time, add -w false to the command.

  The browser should open and close automatically, and the results will be displayed to console.

- If you want code coverage for your tests, add --code-coverage to the command.

  Code coverage information will sit in the coverage subdirectory by default. If you want to change it, you can update the .angular-cli.json file.

Run Integration Tests

To run unit tests, use the ng e2e command. This command compiles the application, launches a development server, and launches a Chrome to run the end-to-end/integration tests. As soon as the tests are done, it shuts down. The results of the tests are displayed to console…

```
$ ng e2e
** NG Live Development Server is listening on localhost:49152, open your
browser on http://localhost:49152/ **
Date: 2017-12-14T01:22:44.223Z
Hash: 020f6d453fbe4177292e
Time: 5659ms
chunk {inline} inline.bundle.js, inline.bundle.js.map (inline) 5.83 kB
[entry] [rendered]
chunk {main} main.bundle.js, main.bundle.js.map (main) 8.47 kB [initial]
```

```
[rendered]
chunk {polyfills} polyfills.bundle.js, polyfills.bundle.js.map (polyfills)
199 kB [initial] [rendered]
chunk {styles} styles.bundle.js, styles.bundle.js.map (styles) 11.4 kB
[initial] [rendered]
chunk {vendor} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.66 MB
[initial] [rendered]
(node:13938) [DEP0022] DeprecationWarning: os.tmpDir() is deprecated. Use
os.tmpdir() instead.

webpack: Compiled successfully.
[17:22:44] I/update - chromedriver: file exists
/home/user/myApp/myApp/node_modules/protractor/node_modules/webdriver-manag
er/selenium/chromedriver_2.34.zip
[17:22:44] I/update - chromedriver: unzipping chromedriver_2.34.zip
[17:22:44] I/update - chromedriver: setting permissions to 0755 for
/home/user/myApp/myApp/node_modules/protractor/node_modules/webdriver-manag
er/selenium/chromedriver_2.34
[17:22:44] I/update - chromedriver: chromedriver_2.34 up to date
[17:22:44] I/launcher - Running 1 instances of WebDriver
[17:22:44] I/direct - Using ChromeDriver directly...
Jasmine started

  my-app App
    ✓ should display welcome message

Executed 1 of 1 spec SUCCESS in 0.507 sec.
[17:22:47] I/launcher - 0 instance(s) of WebDriver still running
[17:22:47] I/launcher - chrome #01 passed
```

NOTE: This seems to want to run on Chrome. It WILL NOT RUN without Chrome.

Build Application

To package your application (for production), use the ng build --prod command. This compiles a build of your application, does a bunch of optimizations on it (e.g. minification, tree shaking, etc..), and dumps it into the dist subdirectory…

```
$ ng build --prod
Date: 2017-12-14T01:28:27.369Z
Hash: a10aedb47eb9147556f3
Time: 14520ms
```

```
chunk {0} polyfills.f039bbc7aaddeebcb9aa.bundle.js (polyfills) 60.9 kB
[initial] [rendered]
chunk {1} main.16f6c3800da4657a0cdb.bundle.js (main) 152 kB [initial]
[rendered]
chunk {2} styles.d41d8cd98f00b204e980.bundle.css (styles) 0 bytes [initial]
[rendered]
chunk {3} inline.08e99da8ad862f62237a.bundle.js (inline) 1.45 kB [entry]
[rendered]
```

> **NOTE**: Remember that when you use --prod, you're also going to be pulling in production environment settings specified in the src/environments/environment.prod.ts file vs the normal development environment settings in the src/environments/environment.ts file.
>
> **NOTE**: The build output directory is specified in the apps.outdir property of the .angular-cli.json file.

How is a production build different from a normal build? …

- Environment.prod.ts is used for your application's settings
- Cache-busting on the files generated (via hashes added to filenames)
- Sourcemaps NOT generated
- CSS extracted to files
- Uglification/minification
- Tree-shaking (dead-code removal)
- AOT

Lint Application

To lint your application, use the ng lint command. This passes your code through the TypeScript/Angular linter…

```
$ ng lint
Warning: The 'no-use-before-declare' rule requires type checking


All files pass linting.
```

In certain cases, whatever the linter is complaining about may be automatically fixable. To do this, use run the lint command again with the --fix option.

Best Practices

The following best practices have been summarized from the Angular best practices lesson.

Files and Folders

Filenames should have extensions that describe what type they're for. This is what the Angular CLI does by default when you ask it to generate something for you. The filename should also match up with the item that it contains.

So for example...

- AppModule → app.module.ts
- PayService → app.service.ts
- PayComponent → pay.component.ts / pay.component.css / pay.component.html / etc..
- CurrencyPipe → currency.pipe.ts

Your folder structure should be broken up by application features. For example, imagine an application called store had 5 different features. This is what the folder structure might look like...

- ./store/payment
- ./store/catalog
- ./store/shipping
- ./store/mailing-list
- ./store/live-support

All services/components/modules specific to a feature would live in the respective folder. If a feature folder gets too large, you can always break it up into further subdirectories.

Coding Conventions

Strive for the preferred naming conventions…

- constants should be camelcase instead of uppercase.
- classes should be camelcase starting with an uppercase (this is normal).
- module classes should be suffixed with Module
- component classes should be suffixed with Component
- service classes should be suffixed with Service
- pipe classes should be suffixed with Pipe

  NOTE: They say you should use camelcase instead of uppercase for consts because the IDE will warn you if you try to change it -- as such there's no point in making it uppercase to make it visually distinct.

Keep your third-party imports and internal imports separate. For TypeScript imports, import your third-party components first. Your internal components should come after those (separated from the third-party ones by a space).

Strive for immutability. Opt for creating a new object vs modifying an existing one. This is because in certain cases, Angular's change detection may not pick up what's happening to the object. Your changes may not propagate through Angular's system.

Module Organization

A good model to follow for breaking down your modules is as follows…

App Module → Top-level module that imports everything and sets it up.

Core Module → The core module (there should be only 1 in your application) should contain any shared singleton services shared across the modules. It should also contain all of the components needed specifically for your application's root component.

> **NOTE**: See the Service Organization section for why declaring all services in this module is a good idea.
>
> **NOTE**: The components created in the core module are not components that are shared across modules, but components specific to the root component.

Shared Module → The shared module (there should be only 1 in your application) should contain any shared components, directives, and pipes. For example, an AJAX loader might go here.

Feature Modules → Feature modules (can be many in your application) correspond to the features of your applications.

Component Organization

Component selectors should have a prefix that identifies what feature area they belong to. So for example, if I had a component in the payment feature module of my application, I may prefix it with app-pay-. Prefixing your selectors will help avoid naming collisions with other imported modules.

> **NOTE**: Remember that when you create a new project using the Angular CLI, it adds linting rules to make sure all your selectors start with the same prefix (app by default). So this additional prefix should go after that prefix.

If your template or CSS styling is over 3 lines, extract them into a separate file.

Use Input()/Output() decorators over inputs/outputs field in the Component annotation. Using the Component annotation field makes it visually difficult to sync up which fields are used for input/output, and you can to manually apply any field name changes to the Component decorator.

Move complex logic out to a service. What qualifies as complex logic? Not exactly sure, but data access definitely falls into that category.

Service Organization

Always declare the @Injectable() decorator on your service. @Injectable() allows other services to inject into your service. Angular recommends you apply this to your service even if it doesn't make use of other services.

> **NOTE**: There's also another way to inject… adding the @Inject(TypeHere) decorator on a constructor parameter will inject that service into the paramere. But this is not recommended unless the datatype of what you're injecting is not the token (aka identifier) for the service.

Remember that services declared in a module are registered globally to all modules (via the root injector), but this isn't the case for lazy loaded modules. If a module is lazy loaded, any services it declares won't be registered with the root injector. They will instead be registered with the injector inside the lazy loaded modules. That means that in some cases, you may end up getting duplicate instances of a service where a singleton service is expected.

> **NOTE**: See the Module Organization section above. This is why we create a core module and define all our services in it -- so we don't get problems like this happening. The core module should never be lazy loaded.

Performance

Use the --prod flag when building your code for deployment. It does optimizations and AOT to produce smaller/faster code.

Monitor the size of your bundle using source maps. When building, use --sourcemaps=true to generate sourcemaps for your code. Then, install the source-map-explorer npm package (install it globally) and run it against your main.bundle.js/vendor.js file to figure out how much space stuff is taking in your overall application (both your own stuff and third-party libraries).

Avoid sorting and filtering with pipes. According to Angular, it's slow and prevents "aggressive minification" of your code. Also, pipes are "pure" by default -- meaning that if the contents being fed into the pipe get updated, the pipe doesn’t update. The pipe only updated when the object reference they point to gets changed.

> **NOTE**: You can get around the pureness problem in 2 ways. You can make what the pipe is sorting immutable (see the part of the Coding Conventions section that says you should always strive for immutability) and do a copy-on-change whenever something’s changed. Or, you can make the pipe impure by by setting the pure property in the @Pipe decorator to false. You should avoid making the pipe impure as it has major performance issues.