Angular Basics

JANUARY 1, 2015

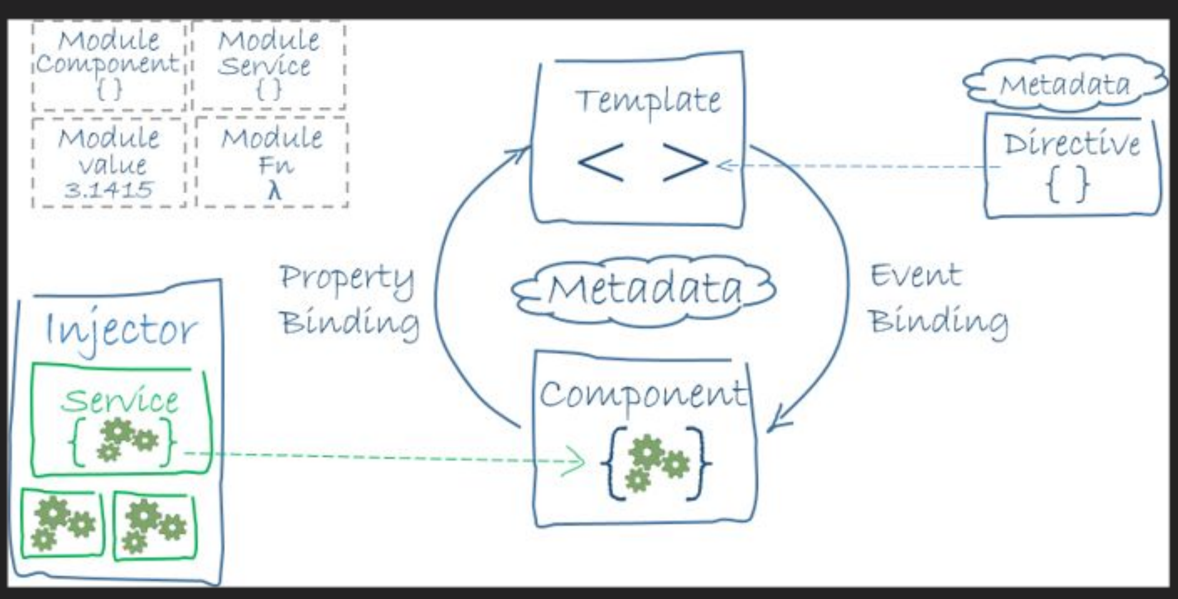
Wiki

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.

1. Install Node.js and npm: https://nodejs.org/en/download/
2. Install the Angular CLI globally: `npm install -g @angular/cli`
3. Generate a new project: `ng new my-app`
4. Go to the project directory: `cd my-app`
5. Launch the server: `ng serve --open`

Angular



Project folder structure

Configuration files:

tsconfig.json - TypeScripts
https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

package.json - npm
https://docs.npmjs.com/files/package.json

.angular-cli.json - Angular CLI configuration
https://github.com/angular/angular-cli/wiki/angular-cli

tslint.json - TSLint (TypeScript code analysis tool)
https://palantir.github.io/tslint/usage/tslint-json/

karma.conf.js - Karma test runner

…

Project folder structure

Folders:

- e2e - automated UI tests (Selenium/Protractor)
- src - our application sources
- node_modules - npm modules (the modules from package.json)

Do not forget to save new npm modules in the package.json file!

How to install/uninstall new node module

Open CMD in the root folder and:

- install all modules in package.json: **npm install**
- install module globally: **npm install *module-name* -g**
- install module locally: **npm install *module-name* --save**
- install dev dependency: **npm install *module-name* --save-dev**
- uninstall: **npm uninstall *module-name* [-g|--save|--save-dev]**
- global modules are stored here: **%appdata%/npm**

NgModule

main.ts - app entry point
app.module.ts - main module

providers	the set of injectable objects that are available in the injector of this module
declarations	a list of directives/pipes that belong to this module
imports	a list of modules whose exported directives/pipes should be available to templates in this module
exports	a list of directives/pipes/modules that can be used within the template of any component that is part of an Angular module that imports this Angular module
entryComponents	a list of components that should be compiled when this module is defined
bootstrap	the components that should be bootstrapped when this module is bootstrapped

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

Generate new module:

ng g module module_name

Component

A component controls a patch of screen called a view. You define a component's application logic—what it does to support the view—inside a class. The class interacts with the view through an API of properties and methods.

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

index.html

```
<body>
      <app-root>Loading...</app-root>
</body>
```

app.component.html

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

The easiest way to display a component property is to bind the property name through ***interpolation***. With interpolation, you put the property name in the view template, enclosed in double curly braces: `{{title}}`.

template expression:

```
<h1>
     Sum of 2 + 2 = {{2+2}}
</h1>
```

you cannot use =. +=, -=, ++, --, new, chaining expressions with ; or ,

Data binding

- interpolation
- property binding
- event binding
- build-in directives

{{value}}

[property] = "value"

(event) = "handler"

[(ng-model)] = "property"

DOM

COMPONENT

HTML attribute vs. DOM property

You deal with DOM properties!

For attribute binding use:

```
<tr>
    <td [attr.colspan]="1 + 1">
     One-Two
    </td>
</tr>
```

Interpolation is a special syntax that Angular converts into a property binding

```
<p><img src="{{heroImageUrl}}"> is the interpolated image.</p>
<p><img [src]="heroImageUrl"> is the property bound image.</p>

<h1>{{title}}</h1>
<h1 [innerHTML]='title'></h1>
```

Event binding

html:

```
<button (click)='onIncrementClick()'>Increment</button>
```

component

```
 onIncrementClick() {
      this.visitors++;
 }
```

$event

html:

```
<button (click)='onIncrementClick($event)'>Increment</button>
```

component

```
onIncrementClick(event: MouseEvent) {
    console.log(`x: ${event.x}, y: ${event.y}`);
    this.visitors++;
}
```

Two-way binding

NgModel - a built-in directive that makes two-way binding easy. It requires *FormsModule*.

```
<input [(ngModel)]='user.name'>
```

NgModel directive only works for an element supported by a ControlValueAccessor that adapts an element to this protocol. Angular provides value accessors for all of the basic HTML form elements.

You don't need a value accessor for an Angular component that you write because you can name the value and event properties to suit Angular's basic two-way binding syntax and skip NgModel altogether.

Create new component: **ng g c component_name** **ng g c user**

```
user.component.ts :    @Input() user: User;

user.component.html : <input [(ngModel)]='user.name'>

app.component.html:   <app-user [(user)]='user'></app-user>
```

Binding syntax: An overview

Data direction	Syntax	Type

One-way
from data source
to view target
	`{{expression}}`
	`[target] = "expression"`
	`bind-target="expression"`
		Interpolation
		Property
		Attribute
		Class
		Style

One-way from view target to data source	`(target)="statement"` `on-target="statement"`	Event
Two-way	`[(target)]="expression"` `bindon-target="expression"`	Two-way

Binding targets

Type	Target	Examples
Property	Element property Component property Directive property	<img [src]=“userImageUrl"> <app-user [user]="currentUser"></app-user> <div [ngClass]="{'special': isSpecial}"></div>
Event	Element event Component event Directive event	<button (click)="onIncrementClick()">Increment</button> <user-detail (deleteRequest)="deleteUser()"></user-detail> <div (myClick)="clicked=$event" clickable>click me</div>
Two-way	Event and property	<input [(ngModel)]="name">
Attribute	Attribute (the exception)	<button [attr.aria-label]="help">help</button>
Class	class property	<div [class.special]="isSpecial">Special</div>
Style	style property	<button [style.color]="isSpecial ? 'red' : 'green'">

Basic Routing

generate components: `ng g c product-list` `ng g c home` `ng g c page-not-found`

app.module:

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

```
const appRoutes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'products', component: ProductListComponent },
    { path: '', redirectTo: '/home', pathMatch: 'full' },
    { path: '**', component: PageNotFoundComponent }
];
```

```
imports: [
RouterModule.forRoot(appRoutes),
```

app.component.html:

```
<nav>
    <a routerLink="/home" routerLinkActive="active">Home</a>
    <a routerLink="/products" routerLinkActive="active">Go to Products</a>
</nav>
<router-outlet></router-outlet>
```

RouterOutlet is a directive, which that acts as placeholder that Angular dynamically fills based on router state.
RouterLink is a directive that lets you link specific parts of your app.

Build-in attribute directives: NgFor & NgIf

NgFor is a repeater directive — a way to present a list of items. You define a block of HTML that defines how a single item should be displayed. You tell Angular to use that block as a template for rendering each item in the list.

```
<div *ngFor='let product of products'>
     <p>{{product.name}}</p>
</div>
```

Take each product in the products array, store it in the local product looping variable, and make it available to the templated HTML for each iteration.

NgIf is a directive that allows to add or remove an element from the DOM

```
<div *ngFor='let product of products'>
     <p>{{product.name}}<span *ngIf='product.price == 345'>- Best Seller!</span></p>
</div>
```

```
<div *ngFor='let product of products'>
     <p>{{product.name}}<span *ngIf='isBestSeller(product)'>- Best
Seller!</span></p>
</div>
```

```
isBestSeller(product) {
     return product.name === 'produc B';
}
```

Lifecycle Hooks

Hook	Purpose and Timing
ngOnChanges()	Respond when Angular (re)sets data-bound input properties. Called before ngOnInit() and whenever one or more data-bound input properties change.
ngOnInit()	Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called once, after the first ngOnChanges().
ngDoCheck()	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after ngOnChanges() and ngOnInit().
ngAfterContentInit()	Respond after Angular projects external content into the component's view. Called once after the first ngDoCheck(). A component-only hook.
ngAfterContentChecked()	Respond after Angular checks the content projected into the component. Called after the ngAfterContentInit() and every subsequent ngDoCheck(). A component-only hook.
ngAfterViewInit()	Respond after Angular initializes the component's views and child views. Called once after the first ngAfterContentChecked(). A component-only hook.
ngAfterViewChecked()	Respond after Angular checks the component's views and child views. Called after the ngAfterViewInit and every subsequent ngAfterContentChecked(). A component-only hook.
ngOnDestroy	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called just before Angular destroys the directive/component.

Component Interaction: @Input decorator

generate new component: `ng g c product`

move product HTML from product-list.component to product.component:

```
<p>{{product.name}}<span *ngIf='isBestSeller(product)'> - Best Seller!</span></p>
```

product-list.component.html:

```
<div *ngFor='let product of products'>
     <app-product [product]='product'></app-product>
</div>
```

product.component.ts:

```
@Input() product: Product;
```

@Input decorator excepts optional alias name to distinguish inputs:

```
@Input('product') product: Product;
```

Component Interaction: @Output decorator

product.component.ts

```
@Output() deleteRequest = new EventEmitter<Product>();

onDeleteClick() {
     this.deleteRequest.emit(this.product);
}
```

product.component.html

```
<button (click)='onDeleteClick()'>Delete</button>
```

product-list.component.ts

```
onProductDelete(product: Product) {
     console.log(`delete request: product name - ${product.name}`);
}
```

product-list.component.html

```
<app-product [product]='product' (deleteRequest)='onProductDelete($event)'></app-product>
```

Pipes

generate pipe: `ng g pipe price`

template: `<p>{{product.price | price}}</p>`

price.pipe.ts:

```
@Pipe({
      name: 'price'
})
export class PricePipe implements PipeTransform {
      transform(price?: number, currencySymbol: string = '$'): any {
         if (!price && price !== 0) {
            return '';
         }
         return `${price} ${currencySymbol}`;
      }
}
```

pass argument: `<p>{{product.price | price:'BYN'}}</p>`

built-in pipes: date, uppercase, lowercase, currency, percent

Services

generate service: **ng g s products**

products.service.ts

```
@Injectable()
export class ProductsService {
    constructor() { }
    getProducts(): Observable<Product[]> {
        return Observable.of([
            <Product>{ name: 'produc A', description: 'pr A desc', price: 123 },
            // ...
        ]);
    }
}
```

product-list.component.ts:

```
constructor(private productsService: ProductsService) { }

ngOnInit() {
    this.productsService.getProducts().subscribe(
        (products: Product[]) => this.products = products,
        (error) => console.log(error)
    );
}
```

```
providers: [ProductsService]
```

Dependency Injection

Angular has a **Hierarchical Dependency Injection** system. There is actually a tree of injectors that parallel an application's component tree.

Injector bubbling - if a dependency is not provided in the component, then Angular requests up the parent component and so on. The requests keep bubbling up until Angular finds an injector that can handle the request or runs out of ancestor injectors. If it runs out of ancestors, Angular throws an error.

@Input decorator marks a class as available to Injector for creation. Injector will throw an error when trying to instantiate a class that does not have @Injectable marker.

Http

products.service.ts:

```
getProducts(): Observable<Product[]> {
    return this.http.get('http://localhost:54145/api/products')
    .map(res => res.json() as Product[]);
}
```

product-list.component.ts:

```
ngOnInit() {
    this.productsService.getProducts().subscribe(
        (products: Product[]) => this.products = products,
        (error) => console.log(error)
    );
}
```

Http + AsyncPipe

products.service.ts:

```
search(term: string): Observable<Product[]> {
    return this.http
        .post(`http://localhost:54145/api/products`, { name: term })
        .map(response => response.json() as Product[]);
}
```

product-list.component.ts:

```
products: Observable<Product[]>;
private searchTerms = new Subject<string>();

// push a search term into the observable stream.
search(term: string): void {
    this.searchTerms.next(term);
}
```

Http + AsyncPipe

```
this.products = this.searchTerms
    .debounceTime(300) // wait 300ms after each keystroke before considering the term
    .distinctUntilChanged() // ignore if next search term is same as previous
    .switchMap(term => term // switch to new observable each time the term changes
        // return the http search observable
        ? this.productsService.search(term)
        // or the observable of empty heroes if there was no search term
        : Observable.of<Product[]>([]))
    .catch(error => {
        // TODO: add real error handling
        console.log(error);
        return Observable.of<Product[]>([]);
});
```

product-list.component.html:

```
<div *ngFor='let product of products | async'>
```