Pipe Checklist: Building a Custom Pipe

Create a class that implements PipeTransform

Write code for the Transform method

Decorate the class with the Pipe decorator

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name: 'spacePipe'
})
export class SpacePipe implements PipeTransform {
  transform(value: string,
            character: string): string { ... }
}
```

Pipe Checklist: Using a Custom Pipe

Add the pipe to the declarations array of an Angular module

```
@NgModule({
  imports: [...],
  declarations: [
      AppComponent,
      ProductListComponent,
      SpacePipe ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Use the pipe in a template

- Pipe character
- Pipe name
- Pipe arguments (separated with colons)

```
{{ product.productCode | spacePipe:'-'}}
```

Checklist: Component as a Directive

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL:
    './product-list.component.html'
})
export class ProductListComponent { }
```

app.component.ts

```
@Component({
 selector: 'pm-root',
 template: `
  <div><h1>{{pageTitle}}</h1>
    <pm-products></pm-products>
  </div>`
})
export class AppComponent { }
```

1

app.module.ts

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

2

Data Binding

Data Binding Checklist: ngModel

product-list.component.html

```
<div class='col-md-4'>
  <input type='text'
     [(ngModel)]='listFilter' />
</div>
```

app.module.ts

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

Data Binding Checklist: Pipes

Pipe character |

Pipe name

Pipe parameters
- Separated with colons

Example

```
{{ product.price | currency:'USD':'symbol':'1.2-2' }}
```

Emitting an Event (@Output)

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: './product-list.component.html'
})
export class ProductListComponent {
  onNotify(message: string): void { }
}
```

star.component.ts

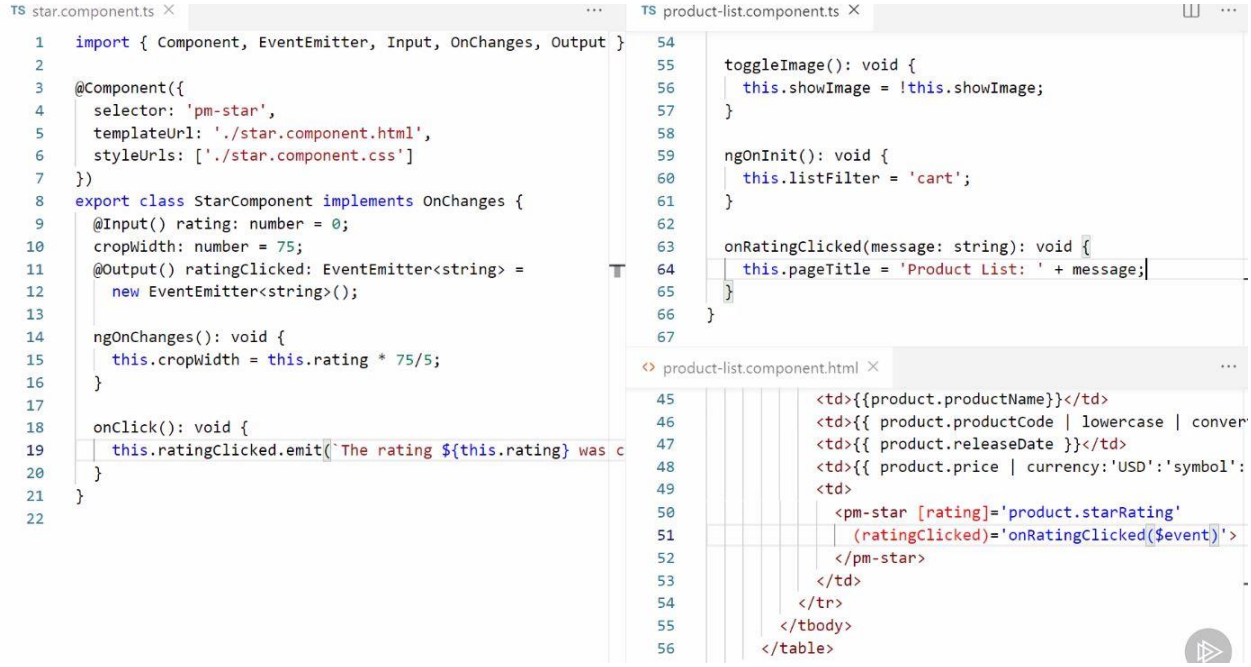
```
@Component({
  selector: 'pm-star',
  templateURL: './star.component.html'
})
export class StarComponent {
 @Input() rating: number;
 cropWidth: number;
 @Output() notify: EventEmitter<string> =
                 new EventEmitter<string>();
 onClick() {
   this.notify.emit('clicked!');
 }
}
```

product-list.component.html

```
<td>
 <pm-star [rating]='product.starRating'
          (notify)='onNotify($event)'>
 </pm-star>
</td>
```

star.component.html

```
<div (click)='onClick()'>
  ... stars ...
</div>
```



What Does an Observable Do?

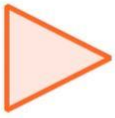
Nothing until we subscribe

`next`: **Next item is emitted**

`error`: **An error occurred and no more items are emitted**

`complete`: **No more items are emitted**

Common Observable Usage



Start the Observable (subscribe)

Pipe emitted items through a set of operators

Process notifications: next, error, complete

Stop the Observable (unsubscribe)

Example

Example

```
import { Observable, range } from 'rxjs';
import { map, filter } from 'rxjs/operators';

const source$: Observable<number> = range(0, 10);

source$.pipe(
   map(x => x * 3),
   filter(x => x % 2 === 0)
).subscribe(x => console.log(x));
```

Result

```
0
6
12
18
24
```

Exception Handling

product.service.ts

```
...
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Observable } from 'rxjs';
import { catchError, tap } from 'rxjs/operators';
...

  getProducts(): Observable<IProduct[]> {
   return this.http.get<IProduct[]>(this.productUrl).pipe(
     tap(data => console.log('All: ', JSON.stringify(data))),
     catchError(this.handleError)
   );
  }

  private handleError(err: HttpErrorResponse) {
  }
```

HTTP Checklist: Subscribing

Call the subscribe method of the returned observable

Provide a function to handle an emitted item

Provide a function to handle any returned errors

```
ngOnInit(): void {
   this.productService.getProducts().subscribe({
      next: products => this.products = products,
      error: err => this.errorMessage = err
   });
}
```

HTTP Checklist: Exception Handling

Add error handling

```
getProducts(): Observable<IProduct[]> {
  return this.http.get<IProduct[]>(this.productUrl).pipe(
    tap(data => console.log(JSON.stringify(data))),
    catchError(this.handleError)
  );
}

private handleError(err: HttpErrorResponse) {
}
```

HTTP Checklist: Calling HTTP Get

Define a dependency for the Http Client Service in the constructor

Create a method for each HTTP request

Call the desired HTTP method, such as get

Use generics to specify the returned type

```
export class ProductService {
  private productUrl = 'www.myService.com/api/products';

  constructor(private http: HttpClient) { }

  getProducts(): Observable<IProduct[]> {
   return this.http.get<IProduct[]>(this.productUrl);
  }
}
```

HTTP Checklist: Unsubscribing

Store the subscription in a variable

```
this.sub = this.ps.getProducts().subscribe(…)
```

Implement the OnDestroy lifecycle hook

```
export class PLComponent implements OnInit, OnDestroy
```

Use the subscription variable to unsubscribe

```
ngOnDestroy(): void {
   this.sub.unsubscribe();
}
```

Injecting the Service

product-list.component.ts

```
...
import { ProductService } from './product.service';

@Component({
  selector: 'pm-products',
  templateUrl: './product-list.component.html'
})
export class ProductListComponent {

 constructor(private productService: ProductService) { }

}
```

OR

product.service.ts

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

product-list.component.ts

```
@Component({
  templateUrl: './product-list.component.html',
  providers: [ProductService]
})
export class ProductListComponent { }
```

below Angular6

app.module.ts

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

Service Checklist: Dependency Injection

Specify the service as a dependency

Use a constructor parameter

Service is injected when component is instantiated

```
constructor(private productService: ProductService) { }
```

Service Checklist: Registering a Service

Select the appropriate level in the hierarchy

- Root application injector if the service is used throughout the application
- Specific component's injector if only that component uses the service

Service Injectable decorator

- Set the `providedIn` property to `'root'`

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

Component decorator

- Set the `providers` property to the service

Service Checklist: Building a Service

Service class

- Clear name
- Use PascalCasing
- Append "Service" to the name
- `export` keyword

Service decorator

- Use Injectable
- Prefix with @; Suffix with ()

Import what we need

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class ProductService {...}
```

Passing Data to a Nested Component (@Input)

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: './product-list.component.html'
})
export class ProductListComponent { }
```

product-list.component.html

```
<td>
 <pm-star [rating]='product.starRating'>
 </pm-star>
</td>
```

star.component.ts

```
@Component({
  selector: 'pm-star',
  templateURL: './star.component.html'
})
export class StarComponent {
  @Input() rating: number;
  cropWidth: number;
}
```

Nesting Checklist: Output Properties

`Output` decorator

Attached to a property declared as an EventEmitter

Use the generic argument to define the event data type

Use the new keyword to create an instance of the EventEmitter

Prefix with @; Suffix with ()

```
export class StarComponent {
 @Output() notify: EventEmitter<string> =
                              new EventEmitter<string>();
}
```

Nesting Checklist: Input Properties

`Input` decorator

Attach to a property of any type

Prefix with @; Suffix with ()

```
export class StarComponent {
 @Input() rating: number;
}
```

Nesting Checklist: Container Component

Use the directive
- Directive name -> nested component's selector

Use property binding to pass data to the nested component

Use event binding to respond to events from the nested component
- Use $event to access the event data passed from the nested component

```
<pm-star [rating]='product.starRating'
         (notify)='onNotify($event)'>
</pm-star>
```

Reading Parameters from a Route

product-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';

    constructor(private route: ActivatedRoute) { }
```

app.module.ts

```
{ path: 'products/:id', component: ProductDetailComponent }
```

Reading Parameters from a Route

Snapshot : Read the parameter one time

```
this.route.snapshot.paramMap.get('id');
```

Observable: Read emitted parameters as they change

```
this.route.paramMap.subscribe(
  params => console.log(params.get('id'))
);
```

Specified string is the route parameter name

```
{ path: 'products/:id',
  component: ProductDetailComponent }
```

Routing Checklist: Passing Parameters

app.module.ts

```
{ path: 'products/:id', component: ProductDetailComponent }
```

product-list.component.html

```
<a [routerLink]="['/products', product.productId]">
   {{product.productName}}
</a>
```

product-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';

constructor(private route: ActivatedRoute) {
   console.log(this.route.snapshot.paramMap.get('id'));
}
```

Routing Checklist: Activate a Route with Code

Use the Router service

- Define it as a dependency

Create a method that calls the navigate method of the Router service

- Pass in the link parameters array

```
import { Router } from '@angular/router';
...
    constructor(private router: Router) { }

    onBack(): void {
      this.router.navigate(['/products']);
    }
```

Add a user interface element

- Use event binding to bind to the created method

```
<button (click)='onBack()'>Back</button>
```

Routing Checklist: Configuring Routes

Add each route to forRoot array
- Order matters

path: Url segment for the route
- No leading slash
- '' for default route
- '**' for wildcard route

component
- Not string name; not enclosed in quotes

```
RouterModule.forRoot([
  { path: 'products', component: ProductListComponent }, …
  { path: '', redirectTo: 'welcome', pathMatch: 'full' },
  { path: '**', redirectTo: 'welcome', pathMatch: 'full' }
])
```

Routing Checklist: Configuring Routes

Define the base element

```
<head>
  ...
  <base href="/">
</head>
```

Add RouterModule

```
@NgModule({
  imports: [ ...,
    RouterModule.forRoot([])
 ],
  declarations: [...],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Routing Checklist: Displaying Components

Nest-able components

- Define a selector

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

- Nest in another component

```
<div><h1>{{pageTitle}}</h1>
  <pm-products></pm-products>
</div>
```

- No route

Routed components

- No selector or nesting
- Configure routes

```
[
 { path: 'products', component: ProductListComponent }
]
```

- Tie routes to actions

Routing Checklist: Placing the View

Add the RouterOutlet directive

- Identifies where to display the routed component's view
- Specified in the host component's template

```
<ul>
  <li><a [routerLink]="['/welcome']">Home</a></li>
  <li><a [routerLink]="['/products']">Product List</a></li>
</ul>
<router-outlet></router-outlet>
```

Protecting Routes with Guards

CanActivate
- Guard navigation to a route

CanDeactivate
- Guard navigation from a route

Resolve
- Pre-fetch data before activating a route

CanLoad
- Prevent asynchronous routing

Routing Checklist: Protecting Routes with Guards

Build a guard service
- Implement the guard type (CanActivate)
- Create the method (canActivate())

Register the guard service provider
- Use the providedIn property

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

@Injectable({ providedIn: 'root' })
export class ProductDetailGuard implements CanActivate {
    canActivate(): boolean { ... }
}
```

Add the guard to the desired route

```
{ path: 'products/:id', canActivate: [ ProductDetailGuard ],
  component: ProductDetailComponent },
```

Angular Module Checklist: NgModule Exports Array

Exports array: Pieces to share

Components, directives, and pipes
- Example: StarComponent

Other modules
- Example: CommonModule, FormsModule

Often only used by shared modules

Angular Module Checklist: NgModule Imports Array

Imports array: Modules this module needs

Modules that provide components, directives, and pipes needed by templates associated with components declared in the module
- Example: CommonModule, FormsModule, SharedModule

Modules that provide system or third-party services
- Example: HttpClientModule
- Imported into AppModule

Feature modules
- Example: ProductModule, InvoiceModule

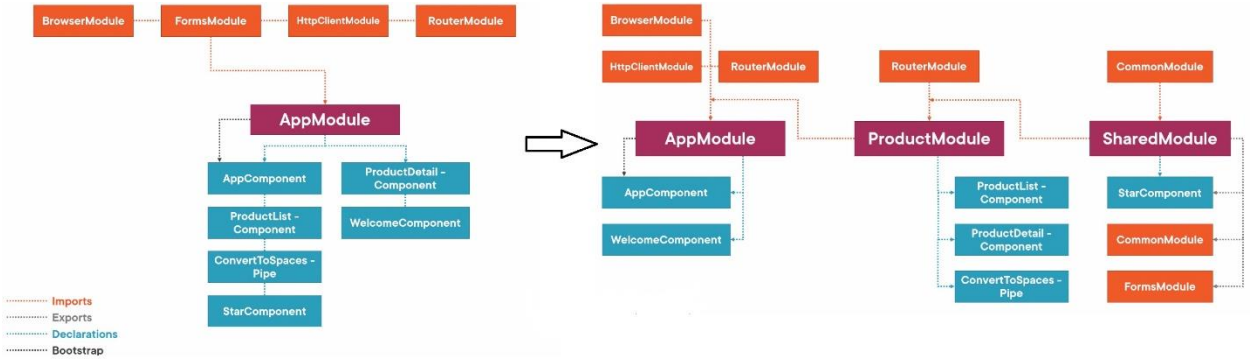
Angular Module Checklist: NgModule Declarations Array

Declarations array: What belongs to this module

Components owned by the module

Declare each component in one and only one module

Directives and pipes used by the declared components



Angular CLI Checklist: Commands

ng help - Displays available commands

ng new - Creates a new Angular application

ng serve - Builds the app and launches a server

ng generate - Generates code

ng add - Adds support for an external library to the app

ng test - Runs unit tests

ng e2e - Runs end-to-end tests*

ng build - Compiles into an output directory

ng deploy - Deploys the application*

ng update - Updates the Angular version for the app

*Requires adding an external package before using the command

Angular CLI Checklist: `ng generate` Commands

class	`ng g cl`
component	`ng g c`
directive	`ng g d`
enum	`ng g e`
guard	`ng g g`
interface	`ng g i`
module	`ng g m`
pipe	`ng g p`
service	`ng g s`