Demo Manual

Table of Contents

Understanding this Manual

Overview

This is the demo manual and is **not provided to attendees until the end of class**. It's purpose is to provide step-by-step directions for instructor demonstrations (demos) used in class.

Attendees do, however, receive the finished code for each demonstration at the beginning of class as part of the course files zip.

It is **recommended that the instructor print a hard copy** of this demo manual and use it as a reference to complete the demonstrations.

Getting Started

1. Open `code\demos\start` as the top level folder in your editor.
2. Open a command-prompt or terminal in `code\demos\start` and run the command.

```
npm install
```

3. Build and start the demo application by running the following command.

```
ng serve -o
```

You can leave `ng serve` running when changing between demos or writing code and it will automatically rebuild your code and reload the browser so you can see your changes.

Reset

After starting a demo you will need to reset the code before beginning the next demo. You can do this by running the following commands.

```
git checkout start -f //checks out the start branch
git clean -df // removes untracked directories and files df
```

For more information see this link about git-clean

External vs. Inline Styles

IMPORTANT

The demos project uses inline templates and inline styles (the html and css is write in the ts file). This is intentional as it greatly helps students see the connection between code in the html and the ts file.

In addition, we have found that when learning...less files equals less confusion.

The official Angular Style Guide recommends to **Extract templates and styles to their own files** (i.e. external templates and styles).

> This recommendation IS followed in the student labs to teach best practices.

> This recommendation is NOT followed in the demos to facilitate learning.

> Reference: Angular Style Guide: Extract templates and styles to their own files
>
> - https://angular.io/guide/styleguide#extract-templates-and-styles-to-their-own-files

Accordingly, this project was created using the following command and it can be useful to explain this before doing the first demo.

```
ng new start --routing --inline-style --inline-template --skip-tests
```

Components

First & Templates (Internal vs. External)

```
ng g component hello-world -d
```

app.component.ts

```
@Component({
  selector: "app-root",
  template: ` <app-hello-world></app-hello-world> `,
  styles: [],
})
export class AppComponent {}
```

create the file hello-world/hello-world.component.html

```
<p>hello-world still works!</p>
```

app.component.ts

```
@Component({
  selector: 'app-hello-world',
+  templateUrl: `./hello-world.component.html`,
  styles: []
})
export class HelloWorldComponent implements OnInit {
  constructor() {}

  ngOnInit() {}
}
```

Nesting

```
ng g component my-button
```

Point out that it was added to app.module.ts declarations

`hello-world/hello-world.component.html`

```
<p>
  hello-world still works!
+ <app-my-button></app-my-button>
</p>
```

Modules

Declarations

```
ng g c tag-one
```

Show how component was added to the `declarations` in the app module.

`app.module.ts`

```
+ import { TagOneComponent } from './tag-one.component';
 ...
@NgModule({
  declarations: [AppComponent,
+               TagOneComponent]
   ...
})
```

```
ng g c tag-one
ng g c tag-two
ng g c tag-three
```

Add new component selectors/tags to the app.component template.

```
@Component({
  selector: "app-root",
  template: `
    <app-tag-one></app-tag-one>
    <app-tag-two></app-tag-two>
    <app-tag-three></app-tag-three>
  `,
  styles: [],
})
export class AppComponent {}
```

Imports & Exports (Feature Modules)

```
ng g module orders
ng g c orders/order-list
ng g c orders/order-detail
ng g c orders/order-form
ng g service orders/shared/order
//note that the service is injected in root we'll talk about that later
ng g pipe orders/shared/order-number
```

import orders module in app module

```
+ import { OrdersModule } from './orders/orders.module';

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

- Exports add `project-detail` to app component template

`app.component.ts`

```
@Component({
  selector: 'app-root',
  template: `
+   <app-order-detail></app-order-detail>
  `,
  styles: []
})
export class AppComponent {}
```

- doesn't work yet

- export project-detail component

`orders\orders.module.ts`

```
@NgModule({
  imports: [CommonModule],
  declarations: [
    OrderListComponent,
    OrderDetailComponent,
    OrderFormComponent,
    OrderNumberPipe
  ],
+  exports: [OrderDetailComponent]
})
export class OrdersModule {}
```

- Now it should render `order detail works`.

Components (continued...)

JSON Pipe

Just start with the solution branch and remove and add back the couple lines showing the `json pipe`.

```
git checkout json-pipe -f
```

```
@Component({
  selector: "app-root",
  template: `
    <div>
      <p>Without JSON pipe:</p>
      <p>{{ object }}</p>
      <p>With JSON pipe (no pre tag):</p>
      <p>{{ object | json }}</p>
      <p>With JSON pipe (and pre tag):</p>
      <pre>{{ object | json }}</pre>
    </div>
  `,
  styles: [],
})
export class AppComponent {
  object: Object = {
    foo: "bar",
    baz: "qux",
    nested: { xyz: 3, numbers: [1, 2, 3, 4, 5] },
  };
}
```

ngFor

```
@Component({
  selector: "app-root",
  template: `
    <ul>
      <li *ngFor="let fruit of fruits">{{ fruit }}</li>
    </ul>
  `,
  styles: [],
})
export class AppComponent {
  fruits = ["Apple", "Orange", "Plum"];
}
```

```
+  <li *ngFor="let fruit of fruits; let i = index;">
     {{i + 1}}.
     {{fruit}}
   </li>
```

The Angular logo used for this demonstration is available in the `src\assets` directory of the start branch.

Interpolation

```
@Component({
  selector: "app-root",
  template: `
    <h2>{{ image.name }}</h2>
    <p>{{ image.path }}</p>
  `,
  styles: [],
})
export class AppComponent {
  image = {
    path: "../assets/angular_solidBlack.png",
    name: "Angular Logo",
  };
}
```

Property Binding

```
@Component({
  selector: "app-root",
  template: `
    <img [src]="image.path" [alt]="image.name" [title]="image.name" />
  `,
  styles: [],
})
export class AppComponent {
  image = {
    path: "../assets/angular_solidBlack.png",
    name: "Angular Logo",
  };
}
```

Input Properties

```
git checkout ngFor -f
```

Generate a list component.

```
ng g component fruit-list
```

```
// fruit-list/fruit-list.component.ts

import { Component, OnInit, Input } from "@angular/core";

@Component({
  selector: "app-fruit-list",
  template: `
    <ul>
      <li *ngFor="let fruit of fruits">{{ fruit }}</li>
    </ul>
  `,
  styles: [],
})
export class FruitListComponent implements OnInit {
  @Input()
  fruits: string[];
  constructor() {}

  ngOnInit() {}
}
```

app.component.ts

```
@Component({
  selector: "app-root",
  template: ` <app-fruit-list [fruits]="data"></app-fruit-list> `,
  styles: [],
})
export class AppComponent {
  data: string[] = ["Apple", "Orange", "Plum"];
}
```

Event Binding

```
@Component({
  selector: "app-root",
  template: `
    <a href="" (click)="onClick($event)">Click Me!</a>
    <p [innerText]="message"></p>
  `,
  styles: [],
})
export class AppComponent {
  message = "";

  onClick(event) {
    event.preventDefault();
    this.message = "clicked";
  }
}
```

Pipes

```
git checkout pipes -f
```

```
@Component({
  selector: "app-root",
  template: `
    <table>
      <tr>
        <th>Pipe Expression</th>
        <th>Formatted Output</th>
      </tr>
      <tr>
        <td>amount | currency: 'USD': "symbol": '2.1-2'</td>
        <td>{{ amount | currency: "USD":"symbol":"2.1-2" }}</td>
      </tr>
      <tr>
        <td>releaseDate | date : 'MM/dd/yyyy'</td>
        <td>{{ releaseDate | date: "MM/dd/yyyy" }}</td>
      </tr>
      <tr>
        <td>amount | number: '3.3-4'</td>
        <td>{{ amount | number: "3.3-4" }}</td>
      </tr>
      <tr>
        <td>percentOfGross | percent: '2.2'</td>
        <td>{{ percentOfGross | percent: "2.2" }}</td>
      </tr>
    </table>
  `,
  styles: [],
})
export class AppComponent {
  amount = 47.341;
  releaseDate: Date = new Date(1975, 4, 25);
  percentOfGross = 0.3245;
}
```

Output Events

```
ng g component email-subscribe
```

```
// email-subscribe.ts

import { Component, OnInit, Output, EventEmitter } from "@angular/core";

@Component({
  selector: "app-email-subscribe",
  template: `
    <input type="text" #email placeholder="email" />
    <button (click)="onClick(email.value)">Subscribe</button>
  `,
  styles: [],
})
export class EmailSubscribeComponent implements OnInit {
  @Output()
  subscribe = new EventEmitter<string>();
  constructor() {}

  ngOnInit() {}
  onClick(email: string) {
    this.subscribe.emit(email);
  }
}
```

```
// app.component.ts
import { Component } from "@angular/core";

@Component({
  selector: "app-root",
  template: `
    <app-email-subscribe
      (subscribe)="onSubscribe($event)"
    ></app-email-subscribe>
    {{ message }}
  `,
  styles: [],
})
export class AppComponent {
  message: string;
```

```
  onSubscribe(email) {
    this.message = `Successfully subscribed. Please check your email ${email}
and click link.`;
  }
}
```

Component Styling

- Inline `app.component.ts`

```
styles: [
  `
    h1 {
      color: rgb(255, 165, 0);
    }
  `,
];
```

- External `app.component.ts`

```
@Component({
  selector: "app-root",
  template: ` <h1>Welcome to {{ title }}!</h1> `,
  styleUrls: ["./app.component.css"],
})
export class AppComponent {
  title = "playground";
}
```

`app.component.css`

```
h1 {
  color: rgb(255, 165, 0);
}
```

ngIf

```
@Component({
  selector: "app-root",
  template: `
    <div>
      <input type="text" placeholder="username" />
      <input type="text" placeholder="password" />
      <button (click)="signIn()">Sign In</button>
    </div>

    <div>Welcome back friend.</div>
  `,
  styles: [],
})
export class AppComponent {
  isSignedIn = false;

  signIn() {
    this.isSignedIn = true;
  }
}
```

```
  <div
+  *ngIf="isSignedIn">
  Welcome back friend.
  </div>
```

```
  <div
+  *ngIf="!isSignedIn">
  <input type="text" placeholder="username">
  ...
```

- ngIf; else

```
@Component({
  selector: 'app-root',
  template: `
  <div
+ *ngIf="!isSignedIn; else signedIn">
  <input type="text" placeholder="username">
  <input type="text" placeholder="password">
  <button (click)="signIn()">Sign In</button>
  </div>

+ <ng-template #signedIn>
  <div>
  Welcome back friend.
  </div>
+ </ng-template>
  `,
  styles: []
})
```

ngSwitch

```
git checkout ngFor -f
```

```
<div [ngSwitch]="fruits.length">
  <p *ngSwitchCase="0">No records returned.</p>
  <p *ngSwitchCase="1">1 record returned.</p>
  <p *ngSwitchDefault>{{fruits.length}} records were returned.</p>
</div>
```

Forms

Reactive Forms Binding

app.module.ts

```
+ import { ReactiveFormsModule } from '@angular/forms';

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

> Note: You may have to manually type the import

> It is easier to understand to type the component code first, then the template.

app.component.ts

```
import { Component, OnInit } from "@angular/core";
import { FormGroup, FormControl } from "@angular/forms";
@Component({
  selector: "app-root",
  template: `
    <h1>Forms</h1>
    <form [formGroup]="loginForm" (submit)="onSubmit()">
      <input formControlName="username" type="text" name="username" /> <br />
      <input formControlName="password" type="text" name="password" /> <br />
      <button>Sign In</button>
    </form>
    <pre>
    {{ loginForm.value | json }}
  </pre
    >
  `,
  styles: [],
})
export class AppComponent implements OnInit {
  loginForm: FormGroup;
  ngOnInit(): void {
    this.loginForm = new FormGroup({
      username: new FormControl(),
      password: new FormControl(),
    });
  }
  onSubmit() {
    console.log(this.loginForm.value);
  }
}
```

Reactive Forms Validation

app.component.ts

```
export class AppComponent implements OnInit {
loginForm: FormGroup;
ngOnInit(): void {
  this.loginForm = new FormGroup(
    {
      username: new FormControl(null,
+ Validators.required),
      password: new FormControl()
    });
}
onSubmit() {
  console.log(this.loginForm.value);
}
}
```

app.component.html

```
 ...
<pre *ngIf="loginForm.get('username').invalid">
{{loginForm.get('username').errors | json}}
</pre>
```

1. See validation errors on refresh.
2. Enter username.
3. Error does not display.
4. Delete username see error displayed again.

UpdateOn

app.component.ts

```
  export class AppComponent implements OnInit {
  loginForm: FormGroup;
  ngOnInit(): void {
    this.loginForm = new FormGroup(
      {
        username: new FormControl(null, Validators.required),
        password: new FormControl()
      },
+     { updateOn: 'blur' }
    );
  }
  onSubmit() {
    console.log(this.loginForm.value);
  }
  }
```

Reactive Forms Validation Messages

`app.component.html`

```
 ...
<div
  *ngIf="loginForm.get('username').dirty &&
loginForm.get('username').invalid &&
loginForm.get('username').touched"
>
  <div *ngIf="loginForm.get('username').hasError('required')">
    Username is required.
  </div>
</div>
```

> Creating the inner `div` first and then wrapping with the outer `div` can facilitate understanding.

Follow these steps to display the validation message. This is intentional but can be confusing.

- Enter some text in the email field
- Delete the text
- Tab out of the email input or cause the email input to lose focus in some way

Reactive Forms Custom Validator

Validate password doesn't contain the phrase password "password".

- Start with this branch

```
git checkout reactive-forms-validation
```

- Output password validation errors

```
<pre *ngIf="loginForm.get('password').invalid">
 {{loginForm.get('password').errors | json}}
 </pre
>
```

- Create Custom validator

Do this in the `app.component.ts` file to make it easier to follow the code.

```
export class CustomValidators {
  static forbiddenPhrase(control: AbstractControl): ValidationErrors | null {
    if (control.value) {
      if (control.value.toLowerCase() === "password") {
        return { forbiddenPhrase: true };
      }
    }
    return null;
  }

  // create method signature
  // paste existing validation function, convert inner function to arrow by removing name
and adding => before opening curly brace
  // add semi-colon at the end
  // remove hard-coded phrase
  static forbiddenPhraseValidatorFn(phrase: string): ValidatorFn {
    return (control: AbstractControl): ValidationErrors | null => {
      if (control.value) {
        if (control.value.toLowerCase() === phrase) {
          return { forbiddenPhrase: true };
        }
      }
```

```
      return null;
    };
  }
}
```

- Add custom validator. app.component.ts

```
import { Component, OnInit } from '@angular/core';
import {
  FormGroup,
  FormControl,
  Validators,
  AbstractControl,
  ValidatorFn,
  ValidationErrors
} from '@angular/forms';
...
export class AppComponent implements OnInit {
  loginForm: FormGroup;
  ngOnInit(): void {
    this.loginForm = new FormGroup(
      {
        username: new FormControl(null, Validators.required,
        password: new FormControl(
          null,
+          CustomValidators.forbiddenPhraseValidatorFn('password')
        )
      }
+      // { updateOn: 'blur' }
    );
  }
  onSubmit() {
    console.log(this.loginForm.value);
  }
}
```

ngClass

```
@Component({
  selector: "app-root",
  template: ` <p>We need to button up our approach out of the loop ... </p> `,
  styles: [
    `
      .highlight {
        background-color: #ffff00;
      }
    `,
  ],
})
export class AppComponent {}
```

```
@Component({
  selector: 'app-root',
  template: `
+    <p (click)="onClick()" [class.highlight]="isHighlighted">
    We need to button ...
    </p>
  `,
  styles: [
    `
      .highlight {
        background-color: #ffff00;
      }
    `
  ]
})
export class AppComponent {
+  isHighlighted = false;
+  onClick() {
+    this.isHighlighted = !this.isHighlighted;
+  }
}
```

```
@Component({
  selector: 'app-root',
  template: `
    <p (click)="onClick()" [class.highlight]="isHighlighted"
+   [class.underline]="true">
    We need to button ...
    </p>
  `,
  styles: [
    `
      .highlight {
        background-color: #ffff00;
      }
+     .underline {
+       text-decoration: underline;
+     }
    `
  ]
})
export class AppComponent {
  isHighlighted = false;
  onClick() {
    this.isHighlighted = !this.isHighlighted;
  }
}
```

```
@Component({
  selector: 'app-root',
  template: `
    <p (click)="onClick()"
+    [ngClass]="calculateClasses()">
    We need to button up ...
    </p>
  `,
  styles: [
    `
      .highlight {
        background-color: #ffff00;
      }
      .underline {
        text-decoration: underline;
      }
    `
  ]
})
```

```
export class AppComponent {
  isHighlighted = false;

  onClick() {
    this.isHighlighted = !this.isHighlighted;
  }

+  calculateClasses() {
+    return {
+      highlight: this.isHighlighted,
+      underline: true
+    };
+ }
}
```

- ngStyle

```
<p (click)="onClick()" [style.background-color]="'orchid'">
  We need to button up ...
</p>
```

```
<p
  [ngStyle]="{'background-color': 'lime',
    'font-size': '20px',
    'font-weight': 'bold'}"
>
  Here we go ...
</p>
```

```
// app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
  <form #signinForm="ngForm" (submit)="onSubmit(signinForm)">
      <input
        type="text"
        placeholder="username"
        ngModel
+        #username="ngModel"
        name="username"
+        required
+        minlength="3"
      /><br />
+      <pre>
+        {{ username.errors | json }}
+      </pre
      >
+      <br />
+      Dirty: {{ username.dirty | json }} <br />
+      Touched: {{ username.touched | json }} <br />
      <input type="text" ngModel name="password" placeholder="password" /><br
/>
      <button>Sign In</button>
    </form>
  `,
  styles: []
})
export class AppComponent {
  ...
}
```

Displaying Validation Messages

```
// app.component.ts

@Component({
  selector: 'app-root',
  template: `
    <form #signinForm="ngForm" (submit)="onSubmit(signinForm)">
      <input
        type="text"
        placeholder="username"
        ngModel
        #username="ngModel"
        name="username"
        required
        minlength="3"
      /><br />
+      <div *ngIf="username.hasError('required') && username.dirty">
+        Username is required.
+      </div>

-      <pre>
-        {{ username.errors | json }}
-      </pre>
-      <br />
-      Dirty: {{ username.dirty | json }} <br />
-      Touched: {{ username.touched | json }} <br />
      <input type="text" ngModel name="password" placeholder="password" /><br
/>

      <button>Sign In</button>
    </form>
  `,
  styles: []
})
export class AppComponent {
 ...
}
```

Services

- Setup

```
git checkout ngFor
```

- Create & Register Service

```
ng g service fruit
```

```
// fruit.service.ts
import { Injectable } from "@angular/core";

@Injectable({
  providedIn: "root",
})
export class FruitService {
  constructor() {}

  list(): string[] {
    return ["Apple", "Orange", "Plum"];
  }
}
```

- Inject Service

```
// app.component.ts
export class AppComponent implements OnInit {
-  fruits = ['Apple', 'Orange', 'Plum'];
+  fruits = [];

+  constructor(private fruitService: FruitService) {}

+  ngOnInit(): void {
+    this.fruits = this.fruitService.list();
+  }
}
```

- With an Observable to handle async

```
// fruit.service.ts
import { Injectable } from "@angular/core";
import { of, Observable } from "rxjs";

@Injectable({
  providedIn: "root",
})
export class FruitService {
  constructor() {}

  list(): Observable<string[]> {
    return of(["Apple", "Orange", "Plum"]);
  }
}
```

```
// app.component.ts
export class AppComponent implements OnInit {
  fruits = [];

  constructor(private fruitService: FruitService) {}

  ngOnInit(): void {
    this.fruitService.list().subscribe((data) ⇒ (this.fruits = data));
  }
}
```

- See Async Happening

```
import { Injectable } from '@angular/core';
import { of, Observable } from 'rxjs';
+ import { delay } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class FruitService {
  constructor() {}

  list(): Observable<string[]> {
    return of(['Apple', 'Orange', 'Plum'])
+     .pipe(delay(4000));
  }
}
```

```
ngOnInit(): void {
    this.fruitService.list().subscribe(data => (this.fruits = data));
+    console.log('completed OnInit');
  }
```

1. Run the application.
2. Open Chrome Devtools.
3. Because we added a 4 second delay, you will see `completed OnInit` logged before the data loads on the page.

RxJS

Observables

Observable: represents the idea of an invokable collection of future values or events.

```
import { Component, OnInit } from '@angular/core';
+ import { of } from 'rxjs';

@Component({
  selector: 'app-root',
  template: ``,
  styles: []
})
export class AppComponent implements OnInit {
+  ngOnInit(): void {
+    const observable$ = of(1, 2, 3);
+    observable$.subscribe(x => console.log(x));
+  }
}
```

OR

```
export class AppComponent implements OnInit {
+  ngOnInit(): void {
+    of(1, 2, 3).subscribe(x => console.log(x));
+  }
}
```

Result

- Open Chrome DevTools
- Switch to the Console tab
- Refresh the browser

```
1
2
3
```

Creating a Stream of DOM Events

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { of,
+ fromEvent } from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
+    <button #myButton>Click Me</button>
  `,
  styles: []
})
export class AppComponent implements OnInit {
  @ViewChild('myButton', { static: true }) button;
  ngOnInit(): void {
+    console.log(this.button);
+    const clicks$ = fromEvent(this.button.nativeElement, 'click');
+    clicks$.subscribe(event => console.log(event));
  }
}
```

Result

- Open Chrome DevTools
- Switch to the Console tab
- Refresh the browser
- Click the button on the page

```
MouseEvent {isTrusted: true, screenX: 30, screenY: 101, clientX: 30, clientY:
22, …}
MouseEvent {isTrusted: true, screenX: 30, screenY: 101, clientX: 30, clientY:
22, …}
...
```

Listening for keyup events

```
import { Component, OnInit, ViewChild } from "@angular/core";
import { fromEvent } from "rxjs";

@Component({
  selector: "app-root",
  template: ` <input #myInput /> `,
  styles: [],
})
export class AppComponent implements OnInit {
  @ViewChild("myInput", { static: true }) input;
  ngOnInit(): void {
    console.log(this.input);
    const keyupEvents$ = fromEvent(this.input.nativeElement, "keyup");
    keyupEvents$.subscribe((event: Event) ⇒
      console.log((event.target as HTMLInputElement).value)
    );
  }
}
```

Result

- Open Chrome DevTools
- Switch to the Console tab
- Refresh the browser
- Enter the letters `abcdef` into the input

```
a
ab
abc
abcd
...
```

Observers

Observer: is a collection of callbacks that knows how to listen to values delivered by the Observable.

```
import { Component, OnInit } from "@angular/core";
import { of, Observer } from "rxjs";

@Component({
  selector: "app-root",
  template: ``,
  styles: [],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    const observable$ = of(1, 2, 3);
    const observer: Observer<any> = {
      next: (x) ⇒ console.log(x),
      complete: () ⇒ console.log("completed"),
      error: (x) ⇒ console.log(x),
    };
    observable$.subscribe(observer);
  }
}
```

Result

- Open Chrome DevTools
- Switch to the Console tab
- Refresh the browser

```
1
2
3
completed
...
```

Subcriptions

Subscription: represents the execution of an Observable, is primarily useful for cancelling the execution.

```
import { Component, OnInit } from '@angular/core';
import { of, Observer,
+ interval } from 'rxjs';

@Component({
  selector: 'app-root',
  template: ``,
  styles: []
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
-    const observable$ = of(1, 2, 3);
+    // Emits ascending numbers, one every second (1000ms)
+   const observable$ = interval(1000);
    const observer: Observer<any> = {
      next: x ⇒ console.log(x),
      complete: () ⇒ console.log('completed'),
      error: x ⇒ console.log(x)
    };
    const subscription = observable$.subscribe(observer);
+   setTimeout(() ⇒ subscription.unsubscribe(), 5000);
  }
}
```

Result

- Open Chrome DevTools
- Switch to the Console tab
- Refresh the browser

```
0
1
2
3
4
...
```

Operators

Continue from prior demo or...

```
git checkout rxjs-subscriptions
```

map

```
import { Component, OnInit } from '@angular/core';
import { of, Observer, interval } from 'rxjs';
+ import { map } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  template: ``,
  styles: []
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    // Emits ascending numbers, one every second (1000ms)
    const observable$ = interval(1000);
    const observer: Observer<any> = {
      next: x => console.log(x),
      complete: () => console.log('completed'),
      error: x => console.log(x)
    };

+    const observableCommingOutOfThePipe$ = observable$.pipe(
+      map(x => x * 10)
+    );
+    const subscription = observableCommingOutOfThePipe$.subscribe(observer);

-   const subscription = observable$.subscribe(observer);
-   setTimeout(() => subscription.unsubscribe(), 5000);
  }
}
```

Result

- Open Chrome DevTools
- Switch to the Console tab
- Refresh the browser

```
0
10
20
30
...
```

tap

```
import { map,
+ tap } from 'rxjs/operators';
 ...

    const observableCommingOutOfThePipe$ = observable$.pipe(
+     tap(x => console.log(x)),
      map(x => x * 10)
    );
```

Result

- Open Chrome DevTools
- Switch to the Console tab
- Refresh the browser

```
0
0
1
10
2
20
3
30
 ...
```

filter

```
import { filter } from 'rxjs/operators';
...

const observableCommingOutOfThePipe$ = observable$.pipe(
  filter(x => x % 2 === 0)
);
```

Result

- Open Chrome DevTools
- Switch to the Console tab
- Refresh the browser

```
0
0
1
10
2
20
3
30
...
```

RxJS code commonly uses the fluent syntax and chains functions.

```
ngOnInit(): void {
    // Emits ascending numbers, one every second (1000ms)
    // const observable$ = interval(1000);
    // const observer: Observer<any> = {
    //   next: x => console.log(x),
    //   complete: () => console.log('completed'),
    //   error: x => console.log(x)
    // };

    // const observableCommingOutOfThePipe$ = observable$.pipe(
    //   filter(x => x % 2 === 0)
    // );

    // observableCommingOutOfThePipe$.subscribe(observer);

    interval(1000)
      .pipe(filter(x => x % 2 === 0))
      .subscribe(x => console.log(x));
  }
```

Subjects

Read & Write

A Subject is like an Observable. It can be subscribed to, just like you normally would with Observables. It also has methods like next(), error() and complete() just like the observer you normally pass to your Observable creation function.

Multicast

The main reason to use Subjects is to multicast. An Observable by default is unicast. Unicasting means that each subscribed observer owns an independent execution of the Observable. To demonstrate this:

Unicast Observable Demo

```
import { Component, OnInit } from "@angular/core";
import { Observable } from "rxjs";

@Component({
  selector: "app-root",
  template: ``,
  styles: [],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    const observable = new Observable((subscriber) => {
      subscriber.next(Math.random());
    });

    // subscription 1
    observable.subscribe((data) => {
      console.log(data); // 0.24957144215097515 (random number)
    });

    // subscription 2
    observable.subscribe((data) => {
      console.log(data); // 0.004617340049055896 (random number)
    });
  }
}
```

Multicast

Subjects can multicast. Multicasting basically means that one Observable execution is shared among multiple subscribers.

Subjects are like EventEmitters, they maintain a registry of many listeners. When calling subscribe on a Subject it does not invoke a new execution that delivers data. It simply registers the given Observer in a list of Observers.

Multicast Subject Example

```
import { Component, OnInit } from "@angular/core";
import { Subject } from "rxjs";

@Component({
  selector: "app-root",
  template: ``,
  styles: [],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    const subject = new Subject();

    // subscription 1
    subject.subscribe((data) => {
      console.log(data); // 0.24957144215097515 (random number)
    });

    // subscription 2
    subject.subscribe((data) => {
      console.log(data); // 0.004617340049055896 (random number)
    });

    subject.next(Math.random());
  }
}
```

Observables = data producers Subjects = data producer and a data consumer

By using Subjects as a data consumer you can use them to convert Observables from unicast to multicast.

Here's a demonstration of that:

```
import { Component, OnInit } from "@angular/core";
import { Subject, Observable } from "rxjs";

@Component({
  selector: "app-root",
  template: ``,
  styles: [],
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    const observable = new Observable((subscriber) => {
      subscriber.next(Math.random());
    });

    const subject = new Subject();

    // subscriber 1
    subject.subscribe((data) => {
      console.log(data);
    });

    // subscriber 2
    subject.subscribe((data) => {
      console.log(data);
    });

    observable.subscribe(subject);
  }
}
```

Practical Example

Search Box

```
import { Component, OnInit } from "@angular/core";
import { Subject } from "rxjs";

@Component({
```

```
  selector: "app-root",
  template: `
    <input
      type="text"
      #term
      (input)="search(term.value)"
      placeholder="search"
    />
    <br />
    <p *ngFor="let message of messages">{{ message }}</p>
  `,
  styles: [],
})
export class AppComponent implements OnInit {
  messages: string[] = [];
  private searchTermStream$ = new Subject<string>();

  ngOnInit(): void {
    this.searchTermStream$.subscribe((term) ⇒
      this.messages.push(`http call for: ${term}`)
    );
  }

  search(term: string) {
    this.searchTermStream$.next(term);
  }
}
```

Result

- Type `angular` quickly in the searchbox
- Ouput is shown below the input on the page

```
http call for: an

http call for: ang

http call for: ang

http call for: angu

http call for: angular

http call for: angular

http call for: angular
...
```

debounceTime

```
import { Component, OnInit } from '@angular/core';
import { Subject } from 'rxjs';
+ import { debounceTime } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  template: `
    <input
      type="text"
      #term
      (keyup)="search(term.value)"
      placeholder="search"
    />
    <br />
    <p *ngFor="let message of messages">{{ message }}</p>
  `,
  styles: []
})
export class AppComponent implements OnInit {
  messages: string[] = [];
  private searchTermStream$ = new Subject<string>();

  ngOnInit(): void {
    this.searchTermStream$
+     .pipe(debounceTime(300))
      .subscribe(term => this.messages.push(`http call for: ${term}`));
  }

  search(term: string) {
    this.searchTermStream$.next(term);
  }
}
```

Result

- Type `angular` quickly in the searchbox
- Ouput is shown below the input on the page

```
http call for: angular
```

distinctUntilChanged

Try

- Type `angular` quickly in the searchbox
- Delete the last letter `r` then retype the `r`
- Ouput is shown below the input on the page

Result

```
http call for: angular

http call for: angula

http call for: angular
```

```
import { Component, OnInit } from '@angular/core';
import { Subject } from 'rxjs';
import { debounceTime,
+ distinctUntilChanged } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  template: `
    <input
      type="text"
      #term
      (keyup)="search(term.value)"
      placeholder="search"
    />
    <br />
    <p *ngFor="let message of messages">{{ message }}</p>
  `,
  styles: []
})
export class AppComponent implements OnInit {
  messages: string[] = [];
  private searchTermStream$ = new Subject<string>();

  ngOnInit(): void {
    this.searchTermStream$
      .pipe(
+       debounceTime(1000),
+      distinctUntilChanged()
      )
      .subscribe(term => this.messages.push(`http call for: ${term}`));
  }

  search(term: string) {
    this.searchTermStream$.next(term);
  }
}
```

Try

- Type `angular` quickly in the searchbox
- Delete the last letter `r` then retype the `r`
- Ouput is shown below the input on the page

Result

```
http call for: angular
```

Notice that I increased the debounceTime so that it's easier to retype the letters you removed.

switchMap

Cancels the orginal obervable and returns a new one

```
this.searchTermStream$
      .pipe(
        debounceTime(1000),
        distinctUntilChanged(),
+        switchMap((term: string) => {
+          return of(`new observable: ${term}`);
+        })
      )
      .subscribe(term => this.messages.push(` ${term}`));
```

Additional Reading

- Understanding RxJS BehaviorSubject, ReplaySubject and AsyncSubject
- Persist Login Status with Behavior Subject

Http

Get

- Checkout start branch

```
git checkout http-start
```

Which has db.json, api script, and json-server installed

- Create model

```
ng g class photo
```

Rename `photo.ts` to `photo.model.ts`

```
export class Photo {
  id: number;
  title: string;
  url: string;
  thumbnailUrl: string;
}
```

- Create service

```
ng g service photo
```

- Import HttpClientModule

```
// app.module.ts
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, AppRoutingModule,
+    HttpClientModule],
```

```
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

- Inject HttpClient service
- Implement getAll method

```
import { Injectable } from '@angular/core';
+ import { HttpClient } from '@angular/common/http';
+ import { Photo } from './photo.model';
+ import { Observable } from 'rxjs';

@Injectable({
+  providedIn: 'root'
})
export class PhotoService {
+  constructor(private http: HttpClient) {}

+  getAll(): Observable<Photo[]> {
+    return this.http.get<Photo[]>('http://localhost:3000/photos');
+  }
}
```

```
// app.component.ts
import { Component, OnInit } from "@angular/core";
import { PhotoService } from "./photo.service";

@Component({
  selector: "app-root",
  template: `
    <h1>Photos</h1>
    <div *ngFor="let photo of photos">
      <img [src]="photo.thumbnailUrl" alt="" />
      <p>{{ photo.title }}</p>
    </div>
  `,
  styles: [],
})
export class AppComponent implements OnInit {
  pphotos: Photo[];
  constructor(private photoService: PhotoService) {}
  ngOnInit(): void {
    this.photoService.getAll().subscribe((data) => (this.photos = data));
  }
}
```

Error Handling

```
// photo.service.ts

import { Injectable } from "@angular/core";
import { HttpClient, HttpErrorResponse } from "@angular/common/http";
import { Photo } from "./photo.model";
import { Observable, throwError } from "rxjs";
import { catchError } from "rxjs/operators";

@Injectable({
  providedIn: "root",
})
export class PhotoService {
  constructor(private http: HttpClient) {}

  getAll(): Observable<Photo[]> {
    return this.http.get<Photo[]>("http://localhost:3000/photos/wrong").pipe(
      catchError((error: HttpErrorResponse) => {
        console.log(error);
        return throwError("An error occured loading the photos.");
      })
    );
  }
}
```

Retry

```
git checkout http-retry -f
```

```
// see photo.service.ts
```

Routing

Routing Basics

- Start Server

```
ng serve -o
```

- Generate Components

```
ng g component home
ng g component about
ng g component contact
```

- Add Routes
  - Snippets
    - a-route-path-eager
    - a-route-path-default

```
const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'home' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
];
```

- Add Navigation and the `router-outlet`

app.component.ts

```
<nav>
  <a [routerLink]="['/home']">Home</a> |
  <a [routerLink]="['/about']">About</a> |
  <a [routerLink]="['/contact']">Contact</a>
</nav>
<router-outlet></router-outlet>
```

- Highlight Active Navigation Item

```
<nav>
  <a routerLinkActive="active" [routerLink]="['/home']">Home</a> |
  <a routerLinkActive="active" [routerLink]="['/about']">About</a> |
  <a routerLinkActive="active" [routerLink]="['/contact']">Contact</a>
</nav>
<router-outlet></router-outlet>
```

Routing Navigation

- Start with routing-navigation-start branch as shown below.
- Merge the service branch.

```
git checkout routing-navigation-start
```

Code Review

All the steps in this next section have already been completed so just review/walkthrough the existing code before moving to the next section.

- Create a movies module.

```
ng g module movies --routing --module=app
```

- Create a movie model class.

```
ng g class movies/shared/movie
```

1. Rename file `movie.ts` to `movie.model.ts`.
2. Add properties to the movie model.

```
export class Movie {
  constructor(
    public id: number,
    public name: string,
    public description: string
  ) {}
}
```

- Create mock movie data.
    1. Create file `movies/shared/mock-movies.ts`
    2. Add these movies.

```
import { Movie } from "./movie.model";

export const MOVIES: Movie[] = [
  new Movie(
    1,
    " Titanic",
    "A seventeen-year-old aristocrat falls in love with a kind but poor
artist aboard the luxurious, ill-fated R.M.S. Titanic."
  ),
  new Movie(
    2,
    " E.T. the Extra-Terrestrial",
    "A troubled child summons the courage to help a friendly alien escape
Earth and return to his home world."
  ),
  new Movie(
    3,
    "The Wizard of Oz",
    "Dorothy Gale is swept away from a farm in Kansas to a magical land of Oz
in a tornado and embarks on a quest with her new friends to see the Wizard
who can help her return home in Kansas and help her friends as well."
  ),
  new Movie(
    4,
    "Star Wars: Episode IV - A New Hope ",
    "Luke Skywalker joins forces with a Jedi Knight, a cocky pilot, a Wookiee
and two droids to save the galaxy from the Empire/`s world-destroying battle-
station while also attempting to rescue Princess Leia from the evil Darth
Vader."
  ),
];
```

- Create a movie service.

```
ng g service movies/shared/movie
```

- Add find and list methods and bring in the needed imports.

movies\shared\movie.service.ts

```
import { of } from "rxjs";
import { Observable } from "rxjs";
import { Injectable } from "@angular/core";

@Injectable()
export class MovieService {
  list(): Observable<Movie[]> {
    return of(MOVIES);
  }

  find(id: number): Observable<Movie> {
    const movie = MOVIES.find((m) => m.id === id);
    return of(movie);
  }
}
```

- Also review the movie components code for the list and detail.
    - list
    - detail

These components were generated with the following commands:

```
ng g component movies/movie-list
```

```
ng g component movies/movie-detail
```

Routing Changes

The steps from here are changes that should be made to the code.

```
//movies/movie-list/movie-list.component.ts

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

@Component({
  selector: 'app-movie-list',
  template: `
  <div>
  <ul>
      <li *ngFor="let movie of movies" >
          <a
+          [routerLink]="['detail', movie.id]">
          {{movie.name}}
          </a>
      </li>
  </ul>
  </div>
  `,
  styles: []
})
export class MovieListComponent implements OnInit {
  movies: Movie[];

  constructor(private movieService: MovieService) {}

  ngOnInit() {
    this.movieService.list().subscribe(data => (this.movies = data));
  }
}
```

- detail

```
//movies/movie-detail/movie-detail.component.ts
...
+ import { MovieService } from '../shared/movie.service';
+ import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-movie-detail',
  template: `
    <div *ngIf="movie">
      <h5>{{ movie.name }}</h5>
      <p>{{ movie.description }}</p>
    </div>
  `,
  styles: []
})
export class MovieDetailComponent implements OnInit {
  movie: Movie;

  constructor(
+    private movieService: MovieService,
+    private route: ActivatedRoute
  ) {}

+  ngOnInit() {
+    this.route.paramMap.subscribe(params => {
+      const id = parseInt(params.get('id'), 10);
+      this.movieService.find(id).subscribe(m => (this.movie = m));
+    });
  }
  }
}
```

- Configure routes.

```
//movies/movies-routing.module.ts
const routes: Routes = [
  { path: 'movies',component: MovieListComponent},
+ { path: 'movies/detail/:id', component: MovieDetailComponent }
];
```

- Add Navigation Item

```
<nav>
<a routerLinkActive="active" [routerLink]="['/home']">Home</a> |
<a routerLinkActive="active" [routerLink]="['/about']">About</a> |
<a routerLinkActive="active" [routerLink]="['/contact']">Contact</a> |
+  <a routerLinkActive="active" [routerLink]="['/movies']">Movies</a>
</nav>
```

Routing (Advanced)

Child Routes

Continue from the previous code (routing-navigation).

- Configure child routes.

```
// movies/movies-routing.module.ts
const routes: Routes = [
  {
    path: 'movies',
    component: MovieListComponent
+  children: [{ path: 'detail/:id', component: MovieDetailComponent }]
  },
-  { path: 'movies/detail/:id', component: MovieDetailComponent }
];
```

- Add an outlet

```
// movies/movie-list/movie-list.component.ts
@Component({
  selector: 'app-movie-list',
  template: `
  <div>
  <ul>
      <li *ngFor="let movie of movies" >
          <a [routerLink]="['detail', movie.id]">{{movie.name}}</a>
      </li>
  </ul>
  </div>
+  <div>
+    <router-outlet></router-outlet>
+  </div>
  `,
  styles: []
})
export class MovieListComponent implements OnInit {
...
```

Lazy Loading

- Configure Movies Module to lazy load.

> It can be useful to look at the size of `main.js` in the Chrome DevTools Network tab and record it on the board before doing this demonstration. Be sure to clear the browser cache so you get the correct file size. You can then repeat the process at the end of the lab and show that the file size is about half of the original size.

Continue from the code from the previous step (routing-navigation).

> Be sure to remove the child route and go back to a traditional route as shown in the step below or other code would need to be adjusted to demonstrate lazy-loading.

```
// movies/movies-routing.module.ts
const routes: Routes = [
  {
-    path: 'movies'
+    path: '',
    component: MovieListComponent,
+    children: [{ path: 'detail/:id', component: MovieDetailComponent }]
  }
-  // { path: 'movies/detail/:id', component: MovieDetailComponent }
];
```

```
// app.module.ts

  imports: [
    BrowserModule,
    AppRoutingModule,
-   MoviesModule
  ],
```

```
// app-routing.module.ts
const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'home' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
+  {
+    path: 'movies',
+    loadChildren: () => import('../app/movies/movies.module').then(m =>
m.MoviesModule)
+  }
];
```

> Note the `import` syntax shown above is new as of Angular 8. For older versions use: `loadChildren: '../app/movies/movies.module#MoviesModule'`. The older "string" syntax will not work in Angular version 9 or higher because it was removed in that version.

Components (Advanced)

Change Detection

Checkout

```
git checkout change-detection -f
```

The following components are already created.

```
parent
child-a
child-b
grandchild-a
```

ChangeDetectionStrategy.Default

Click on each of the buttons to see what components are checked by change detection. Refreshing the page after each button click makes this easier to see.

- If an input "changes" (clicking the parent button which sets the nickname) then the entire tree is checked.
- If an event triggered then the entire tree is checked.

ChangeDetectionStrategy.OnPush

Go into each of the following child components.

- child-a
- child-b
- grandchild-a

Uncomment the line to modify the change detection strategy from:

- the default aptly named `ChangeDetectionStrategy.Default`
- to `ChangeDetectionStrategy.OnPush`

Click on each of the buttons to see what components are checked by change detection. Refreshing the page after each button click makes this easier to see.

- If a component has a changed input or raises an event, causes check on itself (component) and all ancestors
  - If an input "changes" then checks itself (component with input) and all ancestors
  - If an event happens then checks itself (component where event was raised, Ex. click) and all ancestors

Appendixes

Optional demos if class requests and time permits.

Redux (NgRx)

Redux (NgRx) Counter

Documentation available at: https://ngrx.io/

Setup

1. Open: `demos\start`
2. Run `npm install` if you haven't earlier in the class.

Installation

1. Install @ngrx/schematics from npm:

```
ng add @ngrx/schematics
```

2. This will ask you if you want to make the @ngrx/schematics the default collection in your Angular CLI project. Choose `n` for no in this demo.

> NgRx Schematics helps you avoid writing common boilerplate and instead focus on building your application The @ngrx/schematics command prefix is only needed when the default collection isn't set. For example, the command `ng g @ngrx/schematics:action Counter` could just be `ng g action Counter`

3. After installing @ngrx/schematics, install the NgRx dependencies.

```
ng add @ngrx/store
ng add @ngrx/store-devtools
```

In more complex applications, you will also need to install the follow NgRx libraries but there is no need to run the below commands in this example.

```
ng add @ngrx/effects
ng add @ngrx/entity
```

Actions

Create actions.

1. Create the directories `src\app\counter\` and `src\app\counter\shared`.
2. Create a file named `src\app\counter\shared\counter.actions.ts`
3. Describe the counter actions to increment, decrement, and reset its value.

```
// src\app\counter\shared\counter.actions.ts
import { createAction } from "@ngrx/store";

export const increment = createAction("[Counter] Increment");
export const decrement = createAction("[Counter] Decrement");
export const reset = createAction("[Counter] Reset");
```

Reducer

Create a reducer.

1. Create a file named `src\app\counter\shared\counter.reducer.ts`

   Define a reducer function to handle changes in the counter value based on the provided actions.

```
// src\app\counter\shared\counter.reducer.ts
import { createReducer, on } from "@ngrx/store";
import { increment, decrement, reset } from "./counter.actions";

export const initialState = 0;

const _counterReducer = createReducer(
  initialState,
  on(increment, (state) ⇒ state + 1),
  on(decrement, (state) ⇒ state - 1),
  on(reset, (state) ⇒ 0)
);

export function counterReducer(state, action) {
  return _counterReducer(state, action);
}
```

2. Add the count to the state interface and the reducers object and set the counterReducer to manage the state of the counter.

```
// src/app/reducers/index.ts

import {
ActionReducer,
ActionReducerMap,
createFeatureSelector,
createSelector,
MetaReducer
} from '@ngrx/store';
import { environment } from '../../environments/environment';
+ import { counterReducer } from '../counter/shared/counter.reducer';

export interface State {
+  count: number;
}

export const reducers: ActionReducerMap<State> = {
+   count: counterReducer
};

export const metaReducers: MetaReducer<State>[] =
!environment.production
? []
: [];
```

User Interface

1. Generate a counter module.

```
ng g m counter --module=app
```

2. Generate a component to display the counter.

```
ng generate component counter/my-counter
```

3. Update the `MyCounterComponent` class with a selector for the `count`, and methods to dispatch the Increment, Decrement, and Reset actions.

4. Then update the `MyCounterComponent` template with buttons to call the increment, decrement, and reset methods. Use the async pipe to subscribe to the count$ observable.

```
// src/app/counter/my-counter/my-counter.component.ts
import { Component, OnInit
+ , ChangeDetectionStrategy
 } from '@angular/core';
import { Store,
+ select
} from '@ngrx/store';
+ import { Observable } from 'rxjs';
+ import { increment, decrement, reset } from
'../shared/counter.actions';

@Component({
selector: 'app-my-counter',
template: `
+    <button (click)="increment()">Increment</button>
+
+    <div>Current Count: {{ count$ | async }}</div>
+
+    <button (click)="decrement()">Decrement</button>
+
+   <button (click)="reset()">Reset Counter</button>
`,
+  changeDetection: ChangeDetectionStrategy.OnPush,
```

```
styles: []
})
export class MyCounterComponent {
+  count$: Observable<number>;

+  constructor(private store: Store<{ count: number }>) {
+    this.count$ = store.pipe(select('count'));
+  }

+  increment() {
+    this.store.dispatch(increment());
+  }

+  decrement() {
+    this.store.dispatch(decrement());
+  }

+  reset() {
+    this.store.dispatch(reset());
+  }
}
```

5. Export the `MyCounterComponent` from the `CounterModule`.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MyCounterComponent } from './my-counter/my-counter.component';

@NgModule({
declarations: [MyCounterComponent],
imports: [CommonModule],
+  exports: [MyCounterComponent]
})
export class CounterModule {}
```

6. Add the `MyCounterComponent` to your `AppComponent` template.

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

@Component({
selector: 'app-root',
template: `
+    <app-my-counter></app-my-counter>
```

```
`,
styles: []
})
export class AppComponent {}
```

7. If not already running start the application with the following command:

```
ng serve -o
```

8. Open `Chrome DevTools` and demonstrate the time traveling, record replay, and logging features of the `Redux DevTools` extension.

> Directions on installing this extension included as part of the setup document for the class.

Finished code available in `demos\ngrx-counter1`.

LEGACY: Redux (NgRx) Counter

This is the same NgRx Counter example as the previous section. Use these directions that use an older, more verbose syntax (that is still supported) only if the client has an exisiting application where this style of sytax is more common and they want to understand it better.

Setup

```
git checkout start -f
```

Installation

Install @ngrx/schematics from npm:

```
npm install @ngrx/schematics@8 --save-dev
```

> NgRx Schematics helps you avoid writing common boilerplate and instead focus on building your application

After installing @ngrx/schematics, install the NgRx dependencies.

```
npm install @ngrx/{store@8,effects@8,entity@8,store-devtools@8} --save
```

Store

Generate the initial state management and register it within the app.module.ts

```
ng generate @ngrx/schematics:store State --root --module app.module.ts
```

> By adding the StoreModule.forRoot function in the imports array of your AppModule. The StoreModule.forRoot() method registers the global providers needed to access the Store throughout your application.

Actions

Generate a new file named `counter.actions.ts`

```
ng generate @ngrx/schematics:action Counter --flat
```

> The @ngrx/schematics command prefix is only needed when the default collection isn't set.

Describe the counter actions to increment, decrement, and reset its value.

```
// src/app/counter.actions.ts
// delete the generated code and replace with the code below

import { Action } from "@ngrx/store";

export enum ActionTypes {
  Increment = "[Counter Component] Increment",
  Decrement = "[Counter Component] Decrement",
  Reset = "[Counter Component] Reset",
}

export class Increment implements Action {
  readonly type = ActionTypes.Increment;
}

export class Decrement implements Action {
  readonly type = ActionTypes.Decrement;
}

export class Reset implements Action {
  readonly type = ActionTypes.Reset;
}
```

Reducer

Generate a reducer.

```
ng generate @ngrx/schematics:reducer Counter --flat --spec=false
```

Define a reducer function to handle changes in the counter value based on the provided actions.

```
// src/app/counter.reducer.ts
// delete the generated code and replace with the code below

import { Action } from "@ngrx/store";
import { ActionTypes } from "./counter.actions";

export const initialState = 0;

export function counterReducer(state = initialState, action: Action) {
  switch (action.type) {
    case ActionTypes.Increment:
      return state + 1;

    case ActionTypes.Decrement:
      return state - 1;

    case ActionTypes.Reset:
      return 0;

    default:
      return state;
  }
}
```

Add the count to the state interface and the reducers object and set the counterReducer to manage the state of the counter.

```
// src/app/reducers/index.ts

import {
  ActionReducer,
  ActionReducerMap,
  createFeatureSelector,
  createSelector,
  MetaReducer
} from '@ngrx/store';
import { environment } from '../../environments/environment';
+ import { counterReducer } from '../counter.reducer';

export interface State {
+  count: number;
}

export const reducers: ActionReducerMap<State> = {
+   count: counterReducer
};

export const metaReducers: MetaReducer<State>[] = !environment.production
  ? []
  : [];
```

Component

```
ng generate component my-counter --spec=false
```

Update the `MyCounterComponent` class with a selector for the `count`, and methods to dispatch the Increment, Decrement, and Reset actions.

Then, update the `MyCounterComponent` template with buttons to call the increment, decrement, and reset methods. Use the async pipe to subscribe to the count$ observable.

```
// src/app/my-counter/my-counter.component.ts

import { Component } from '@angular/core';
+ import { Store, select } from '@ngrx/store';
+ import { Observable } from 'rxjs';
+ import { Increment, Decrement, Reset } from '../counter.actions';

@Component({
  selector: 'app-my-counter',
   template: `
+    <div>Current Count: {{ count$ | async }}</div>
+
+    <button (click)="increment()">Increment</button>
+
+    <button (click)="decrement()">Decrement</button>
+
+    <button (click)="reset()">Reset Counter</button>
  `,
  styleUrls: ['./my-counter.component.css'],
})
export class MyCounterComponent {
+  count$: Observable<number>;
+  ngOnInit(): void {}
+  constructor(private store: Store<{ count: number }>) {
+    this.count$ = store.pipe(select('count'));
+  }

+  increment() {
+    this.store.dispatch(new Increment());
+  }

+  decrement() {
```

```
+    this.store.dispatch(new Decrement());
+  }

+  reset() {
+    this.store.dispatch(new Reset());
+  }
}
```

Add the `MyCounter` component to your `AppComponent` template. Delete the default generated html content.

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

@Component({
  selector: 'app-root',
  template: `
+    <app-my-counter></app-my-counter>
  `,
  styles: []
})
export class AppComponent {}
```

If not already running start the application with the following command

```
ng serve -o
```

Open `Chrome DevTools` and demonstrate the time traveling, record replay, and logging features of the `Redux DevTools` extension.

> Directions on installing this extension included as part of the setup document for the class.

Finished code available in `demos\ngrx-counter`.

NgRx Lab

Setup

1. **Copy** the folder `\code\labs\lab29\complete\project-manage` into your `working` directory.
2. In the `working\project-manage` directory run the command `npm install`.

Installation

1. **Install** `@ngrx/schematics` from npm:

```
ng add @ngrx/schematics
```

2. This will ask you if you want to make the `@ngrx/schematics` the **default collection** in your Angular CLI project. **Choose** `n` for **no** in this lab.

1. After installing @ngrx/schematics, **install** the NgRx **following library** dependencies.

```
ng add @ngrx/store
ng add @ngrx/store-devtools
ng add @ngrx/effects
```

2. **Open** `src\app\app.module.ts` and notice that the `StoreModule` was automatically imported by the `ng add` commands and the Redux Dev Tools are configured.

Actions

1. **Create** the `src\app\projects\shared` and `src\app\projects\shared\state` **directories**.

2. **Create** the **file** `src\app\projects\shared\state\project.actions.ts`.

3. **Add** the following **actions** using the `createAction` helper function.

   `src\app\projects\shared\state\project.actions.ts`

```
import { createAction, props } from "@ngrx/store";
import { Project } from "../project.model";

export const load = createAction("[Project] Load");
export const loadSuccess = createAction(
  "[Project] Load Success",
  props<{ projects: Project[] }>()
);
export const loadFail = createAction(
  "[Project] Load Fail",
  props<{ error: any }>()
);

export const save = createAction(
  "[Project] Save",
  props<{ project: Project }>()
);
export const saveSuccess = createAction(
  "[Project] Save Success",
  props<{ project: Project }>()
);
export const saveFail = createAction(
  "[Project] Save Fail",
  props<{ error: any }>()
);
```

State

1. **Create** the **file** `src\app\projects\shared\state\project.reducer.ts`.
2. Define the state for the slice of the state related to projects including:
    1. an interface
    2. initial or default values
    3. selectors to select different parts of state

src\app\projects\shared\state\project.reducer.ts

```
import { Project } from "../project.model";
import { State } from "src/app/reducers";

export interface ProjectState {
  loading: boolean;
  saving: boolean;
  error: string;
  projects: Project[];
}

export const initialState = {
  loading: false,
  saving: false,
  error: "",
  projects: [],
};

export const getProjects = (state: State) =>
state.projectState.projects;
export const getLoading = (state: State) => state.projectState.loading;
export const getSaving = (state: State) => state.projectState.saving;
export const getError = (state: State) => state.projectState.error;
```

Reducer

1. Create the `projectReducer` using the `createReducer` helper function.

src\app\projects\shared\state\project.reducer.ts

```
...
import { createReducer, on } from '@ngrx/store';
import {
load,
loadSuccess,
loadFail,
save,
saveSuccess,
saveFail
} from './project.actions';

const _projectReducer = createReducer(
initialState,
on(load, state => ({ ...state, loading: true })),
on(loadSuccess, (state, { projects }) => ({
    ...state,
    projects,
    loading: false,
    saving: false
})),
on(loadFail, (state, { error }) => ({ ...state, error, loading: false
})),
on(save, state => ({ ...state, saving: true })),
on(saveSuccess, (state, { project }) => {
    const updatedProjects = state.projects.map(item =>
    project.id === item.id ? project : item
    );
    return {
    ...state,
    projects: updatedProjects,
    saving: false
    };
}),
on(saveFail, (state, { error }) => ({ ...state, error, saving: false }))
);

export function projectReducer(state, action) {
return _projectReducer(state, action);
}
```

Effects

1. Create the `ProjectEffects` class and the `load$` and `save$` effects using the `createEffect` helper function.

   `src\app\projects\shared\state\project.effects.ts`.

```
import { Injectable } from "@angular/core";
import { of } from "rxjs";
import { Actions, ofType, createEffect } from "@ngrx/effects";

import { switchMap, catchError, map, mergeMap } from "rxjs/operators";
import { ProjectService } from "../project.service";
import {
  load,
  loadSuccess,
  loadFail,
  save,
  saveSuccess,
  saveFail,
} from "./project.actions";

@Injectable()
export class ProjectEffects {
  load$ = createEffect(() => {
    return this.actions$.pipe(
      ofType(load),
      switchMap(() => {
        return this.projectService.list().pipe(
          map((data) => loadSuccess({ projects: data })),
          catchError((error) => of(loadFail({ error: error })))
        );
      })
    );
  });

  save$ = createEffect(() => {
    return this.actions$.pipe(
      ofType(save),
      mergeMap(({ project }) => {
        return this.projectService.put(project).pipe(
          map(() => saveSuccess({ project })),
          catchError((error) => of(saveFail({ error: error })))
        );
      })
    );
```

```
    });

    constructor(
      private actions$: Actions,
      private projectService: ProjectService
    ) {}
  }
```

Configure

1. Add the project related state and reducer to the root state and reducer.

   src\app\reducers\index.ts.

```
import {
ActionReducer,
ActionReducerMap,
createFeatureSelector,
createSelector,
MetaReducer
} from '@ngrx/store';
import { environment } from '../../environments/environment';
+ import {
+   ProjectState,
+   projectReducer
+ } from '../projects/shared/state/project.reducer';

export interface State {
+  projectState: ProjectState;
}

export const reducers: ActionReducerMap<State> = {
+  projectState: projectReducer
};

export const metaReducers: MetaReducer<State>[] =
!environment.production
? []
: [];
```

Connect the Container & Store

1. Refactor the `ProjectsContainerComponent` to use the `Store` instead of the `ProjectService` to access data. More specifically:

    1. Change all class members to be Observables.
    2. Inject the `Store` instead of the `ProjectService` in the component's constructor.
    3. Dispatch the `load` action in `ngOnInit` instead of calling the service directly.
    4. Dispatch the `save` action in the `onSaveListItem` event handler method instead of calling the service directly.

src\app\projects\projects-container\projects-container.component.ts

```
import { Component, OnInit
+ , ChangeDetectionStrategy
 } from '@angular/core';
import { Project } from '../shared/project.model';
- import { ProjectService } from '../shared/project.service';
+ import { Observable } from 'rxjs';
+ import { Store, select } from '@ngrx/store';
+ import { State } from 'src/app/reducers';
+ import { load, save } from '../shared/state/project.actions';
+ import {
+   getProjects,
+   getError,
+   getLoading,
+   getSaving
+ } from '../shared/state/project.reducer';

@Component({
selector: 'app-projects-container',
templateUrl: './projects-container.component.html',
styleUrls: ['./projects-container.component.css'],
+ changeDetection: ChangeDetectionStrategy.OnPush
})
export class ProjectsContainerComponent implements OnInit {
-  projects: Project[];
-  errorMessage: string;
-  loading: boolean;
+  projects$: Observable<Project[]>;
+  errorMessage$: Observable<string>;
+  loading$: Observable<boolean>;
+  saving$: Observable<boolean>;
```

```
-  constructor(private projectService: ProjectService) {}
+  constructor(private store: Store<State>) {}

ngOnInit() {
-    this.loading = true;
-    this.projectService.list().subscribe(
-      data => {
-        this.loading = false;
-        this.projects = data;
-      },
-      error => {
-        this.loading = false;
-        this.errorMessage = error;
-      }
-    );
+    this.projects$ = this.store.pipe(select(getProjects));
+    this.errorMessage$ = this.store.pipe(select(getError));
+    this.loading$ = this.store.pipe(select(getLoading));
+    this.saving$ = this.store.pipe(select(getSaving));
+    this.store.dispatch(load());
}

onSaveListItem(event: any) {
    const project: Project = event.item;
-    this.projectService.put(project).subscribe(
-      updatedProject => {
-        const index = this.projects.findIndex(
-          element => element.id === project.id
-        );
-        this.projects[index] = project;
-      },
-      error => (this.errorMessage = error)
-    );
-  }
+   this.store.dispatch(save({ project }));

}
```

src\app\projects\projects-container\projects-container.component.html

```
<h1>Projects</h1>
<div *ngIf="loading$ | async" class="center-page">
  <span class="spinner primary"></span>
  <p>Loading ...</p>
</div>
<span *ngIf="saving$ | async" class="toast"> Saving ... </span>
<div class="row">
  <div *ngIf="errorMessage$ | async as errorMessage" class="card large
error">
    <section>
      <p><span class="icon-alert inverse"></span> {{ errorMessage }}</p>
    </section>
  </div>
</div>
<ng-container *ngIf="projects$ | async as projects">
  <app-project-list
    [projects]="projects"
    (saveListItem)="onSaveListItem($event)"
  >
  </app-project-list>
</ng-container>
```

1. Register the `ProjectEffects`.

src\app\app.module.ts

```
+ import { ProjectEffects } from './projects/shared/state/project.effects';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ProjectsModule,
    HttpClientModule,
    HomeModule,
    StoreModule.forRoot(reducers, {
      metaReducers,
      runtimeChecks: {
        strictStateImmutability: true,
        strictActionImmutability: true
```

```
      }
    }),
    StoreDevtoolsModule.instrument({
      maxAge: 25,
      logOnly: environment.production
    }),
    EffectsModule.forRoot([AppEffects,
+   ProjectEffects])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

1. Verify the application functionality works as it did previously including:
    1. Loading and saving data.
    2. Handles errors appropriately when the backend is down.
    3. Displays loading and saving messages.
2. Lastly, verify you can now time travel.
    1. Open Chrome DevTools (F12 or fn+F12 on Windows)
    2. Install the Redux DevTools Extension if you haven't already.
    3. Use the application and then travel back in time and replay your actions.

NgRx Resources

- Official Documentation
- Example Application
- New Features in NgRx 8
- NgRx Resources

Online courses

Pluralsight (Deborah Kurata and Duncan Hunter) Angular NgRx: Getting Started

Ultimate Angular (Todd Motto) NGRX Store + Effects

Redux & GraphQL Resources

GraphQL

How GraphQL Replaces Redux

Free Video Course on Redux by Creator

Future: Less State Management

You Might Not Need Redux

Presentational vs Container components divide

Template-driven Forms

Template Forms Binding

```
// app.module.ts
+ import { FormsModule } from '@angular/forms';

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

```
// app.component.ts
import { Component } from "@angular/core";

@Component({
  selector: "app-root",
  template: `
    <form #signinForm="ngForm" (submit)="onSubmit(signinForm)">
      <input type="text" ngModel name="username" placeholder="username" /><br
/>
      <input type="text" ngModel name="password" placeholder="password" /><br
/>
      <button>Sign In</button>
    </form>
    <pre>
      {{ signinForm.value | json }}
    </pre
    >
  `,
  styles: [],
})
export class AppComponent {
  onSubmit(form) {
    console.log(form.value);
  }
}
```

Template Forms Validation

```
// app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
  <form #signinForm="ngForm" (submit)="onSubmit(signinForm)">
      <input
        type="text"
        placeholder="username"
        ngModel
+        #username="ngModel"
        name="username"
+        required
+        minlength="3"
      /><br />
+      <pre>
+        {{ username.errors | json }}
+      </pre
      >
+      <br />
+      Dirty: {{ username.dirty | json }} <br />
+      Touched: {{ username.touched | json }} <br />
      <input type="text" ngModel name="password" placeholder="password" /><br
/>
      <button>Sign In</button>
    </form>
  `,
  styles: []
})
export class AppComponent {
  ...
}
```

Displaying Validation Messages

```
// app.component.ts

@Component({
  selector: 'app-root',
  template: `
    <form #signinForm="ngForm" (submit)="onSubmit(signinForm)">
      <input
        type="text"
        placeholder="username"
        ngModel
        #username="ngModel"
        name="username"
        required
        minlength="3"
      /><br />
+      <div *ngIf="username.hasError('required') && username.dirty">
+        Username is required.
+      </div>

-      <pre>
-        {{ username.errors | json }}
-      </pre>
-      <br />
-      Dirty: {{ username.dirty | json }} <br />
-      Touched: {{ username.touched | json }} <br />
      <input type="text" ngModel name="password" placeholder="password" /><br
/>

      <button>Sign In</button>
    </form>
  `,
  styles: []
})
export class AppComponent {
 ...
}
```

Two-way Binding

```
@Component({
  selector: "app-root",
  template: `
    <input
      [value]="message"
      (input)="message = $event.target.value"
      type="text"
    />
    <p>{{ message }}</p>
  `,
  styles: [],
})
export class AppComponent {
  message = "";
}
```

```
import { FormsModule } from "@angular/forms";

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

```
@Component({
  selector: "app-root",
  template: `
    <input [(ngModel)]="message" type="text" />
    <p>{{ message }}</p>
  `,
  styles: [],
})
export class AppComponent {
  message = "";
}
```

Angular Material: Introduction

Installation

1. In a command-prompt (Windows) or terminal (Mac) run the following commands.

```
ng new my-mat-proj
cd my-mat-proj
ng add @angular/material
code .
```

   Choose:

   - indigo-pink.css
   - HammerJS: Yes
   - Browser animations: Y

2. Open `my-mat-proj` in your editor.

```
code .
```

3. Look at the changes using a Git diff.

> What the command does:

```
- Ensure project dependencies are placed in package.json
- Enable the BrowserAnimationsModule your app module
- Add either a prebuilt theme or a custom theme
- Add Roboto fonts to your index.html
- Add the Material Icon font to your index.html
- Add global styles to
- Remove margins from body
- Set height: 100% on html and body
- Make Roboto the default font of your app
- Install and import hammerjs for gesture support in your project
```

3. Add a slider.

src\app\app.module.ts

```
import { MatSliderModule } from '@angular/material/slider';
…
@NgModule ({....
  imports: [ ... ,
  MatSliderModule,
…]
})
```

src\app\app.component.html

```
// delete contents
<mat-slider min="1" max="100" step="1" value="1"></mat-slider>
```

4. Build and run the application.

```
ng serve
```

Navigation

1. Generate a navigation component.

```
ng generate @angular/material:nav mainnav
```

2. Add the component to the main content area.

src/app/mainnav/mainnav.component.html

```
</mat-toolbar>
    <!-- Add Content Here -->
+    <h1>Main Content</h1>
+    <p>This is the main content</p>
  </mat-sidenav-content>
```

> Resize browser smaller to see hamburger menu

3. Make the changes below to show the menu by default.

src/app/mainnav/mainnav.component.html

```
<mat-sidenav-container class="sidenav-container">
  <mat-sidenav
    #drawer
    class="sidenav"
    fixedInViewport
    [attr.role]="(isHandset$ | async) ? 'dialog' : 'navigation'"
    [mode]="(isHandset$ | async) ? 'over' : 'side'"
-    [opened]="(isHandset$ | async) === false"
  >

 ...
<mat-sidenav-content>
    <mat-toolbar color="primary">
      <button
        type="button"
        aria-label="Toggle sidenav"
        mat-icon-button
        (click)="drawer.toggle()"
-        *ngIf="isHandset$ | async"
```

```
        >
          <mat-icon aria-label="Side nav toggle icon">menu</mat-icon>
        </button>
        <span>my-mat-proj</span>
      </mat-toolbar>
```

Dashboard

1. Generate a dashboard component.

```
ng generate @angular/material:dashboard dashboard
```

2. Add the component to the main content area.

src/app/mainnav/mainnav.component.html

```
...
-     <h1>Test</h1>
+     <app-dashboard></app-dashboard>
  </mat-sidenav-content>
</mat-sidenav-container>
```

Table

1. Generate a table component.

```
ng generate @angular/material:table my-table
```

2. Add the component to the main content area.

src/app/mainnav/mainnav.component.html

```
...
    <app-dashboard></app-dashboard>
+     <app-my-table></app-my-table>
  </mat-sidenav-content>
</mat-sidenav-container>
```

Forms

1. Generate an address form component.

```
ng generate @angular/material:address-form address-form
```

2. Add the component to the main content area.

src/app/mainnav/mainnav.component.html

```
...
    <app-dashboard></app-dashboard>
    <app-my-table></app-my-table>
+    <app-address-form></app-address-form>
  </mat-sidenav-content>
</mat-sidenav-container>
```

Reference

- Angular Material: Getting Started
- Angular Material: Schematics

Angular Material: CDK

Angular Material components use the `Component Development Kit` (CDK) library for reusable behavior and combines it with the styles to create components that implement Google's Material Design Specification.

CDK: Installation

Steps

1. Install

```
npm install @angular/cdk@10 faker
```

> Note: `faker` is a JavaScript library used for generating data that we will use for one of the examples

2. Import the module

```
//app.module.ts
import {DragDropModule} from '@angular/cdk/drag-drop';
import { ScrollingModule } from '@angular/cdk/scrolling';

@NgModule({
  imports: [ ... ,DragDropModule, ScrollingModule ]
})
export class AppModule { }
```

CDK: Drag & Drop

(using the `DragDropModule`)

The `@angular/cdk/drag-drop` module provides you with a way to easily and declaratively create drag-and-drop interfaces, with support for free dragging, sorting within a list, transferring items between lists, animations, touch devices, custom drag handles, previews, and placeholders, in addition to horizontal lists and locking along an axis.

Steps

1. Add the following styles

```
/*
src/styles.css
*/

.example-container {
  width: 400px;
  max-width: 100%;
  margin: 0 25px 25px 0;
  display: inline-block;
  vertical-align: top;
}

.example-list {
  border: solid 1px #ccc;
  min-height: 60px;
  background: white;
  border-radius: 4px;
  overflow: hidden;
  display: block;
}

.example-boundary {
  width: 400px;
  height: 400px;
  max-width: 100%;
  border: dotted #ccc 2px;
}

.example-box {
  width: 200px;
  height: 200px;
```

```
  border: solid 1px #ccc;
  color: rgba(0, 0, 0, 0.87);
  cursor: move;
  display: flex;
  justify-content: center;
  align-items: center;
  text-align: center;
  background: #fff;
  border-radius: 4px;
  position: relative;
  z-index: 1;
  transition: box-shadow 200ms cubic-bezier(0, 0, 0.2, 1);
  box-shadow: 0 3px 1px -2px rgba(0, 0, 0, 0.2), 0 2px 2px 0 rgba(0, 0, 0,
0.14),
    0 1px 5px 0 rgba(0, 0, 0, 0.12);
}

.example-box:active {
  box-shadow: 0 5px 5px -3px rgba(0, 0, 0, 0.2), 0 8px 10px 1px rgba(0, 0, 0,
0.14),
    0 3px 14px 2px rgba(0, 0, 0, 0.12);
}

.example-box {
  padding: 20px 10px;
  border-bottom: solid 1px #ccc;
  color: rgba(0, 0, 0, 0.87);
  display: flex;
  flex-direction: row;
  align-items: center;
  justify-content: space-between;
  box-sizing: border-box;
  cursor: move;
  background: white;
  font-size: 14px;
}

.cdk-drag-preview {
  box-sizing: border-box;
  border-radius: 4px;
  box-shadow: 0 5px 5px -3px rgba(0, 0, 0, 0.2), 0 8px 10px 1px rgba(0, 0, 0,
0.14),
    0 3px 14px 2px rgba(0, 0, 0, 0.12);
}

.cdk-drag-placeholder {
  opacity: 0;
}

.cdk-drag-animating {
```

```
  transition: transform 250ms cubic-bezier(0, 0, 0.2, 1);
}

.example-box:last-child {
  border: none;
}

.example-list.cdk-drop-list-dragging .example-box:not(.cdk-drag-placeholder)
{
  transition: transform 250ms cubic-bezier(0, 0, 0.2, 1);
}
```

2. Create a `div` with the `cdKDrag` directive

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
+    <div class="example-box" cdkDrag>
+      Drag me around
+    </div>
  `,
  styles: []
})
export class AppComponent {}
```

3. In the browser you can now drag the div around the page.

4. You can also drag the div off the page.

5. Wrap the another `div` around the draggable `div`. This wrapper element will be the boundary within which you can drag.

```
<div class="example-boundary">
  <div class="example-box" cdkDragBoundary=".example-boundary" cdkDrag>
    Drag me around
  </div>
</div>
```

> Be sure to include the `.` before the class name `example-boundary` in the `cdkDragBoundary` directive.

6. Drag the element and notice that it stays within the bounds of the boundary box.

> You can stop here but if time permits you can type the code below to create a list that reorders itself.

Reorder List

1. Update the code

```
// app.component.ts

import { Component } from "@angular/core";
import { CdkDragDrop, moveItemInArray } from "@angular/cdk/drag-drop";

@Component({
  selector: "app-root",
  template: `
    <div cdkDropList class="example-list"
(cdkDropListDropped)="drop($event)">
      <div class="example-box" *ngFor="let movie of movies" cdkDrag>
        {{ movie }}
      </div>
    </div>
  `,
  styles: [],
})
export class AppComponent {
  movies = [
    "Episode I - The Phantom Menace",
    "Episode II - Attack of the Clones",
    "Episode III - Revenge of the Sith",
    "Episode IV - A New Hope",
    "Episode V - The Empire Strikes Back",
    "Episode VI - Return of the Jedi",
    "Episode VII - The Force Awakens",
    "Episode VIII - The Last Jedi",
  ];

  drop(event: CdkDragDrop<string[]>) {
    moveItemInArray(this.movies, event.previousIndex, event.currentIndex);
  }
}
```

8. Comment out the first `example-box` style as shown below

```
/* src/styles.css */

/* .example-box {
  width: 200px;
```

```
  height: 200px;
  border: solid 1px #ccc;
  color: rgba(0, 0, 0, 0.87);
  cursor: move;
  display: flex;
  justify-content: center;
  align-items: center;
  text-align: center;
  background: #fff;
  border-radius: 4px;
  position: relative;
  z-index: 1;
  transition: box-shadow 200ms cubic-bezier(0, 0, 0.2, 1);
  box-shadow: 0 3px 1px -2px rgba(0, 0, 0, 0.2), 0 2px 2px 0 rgba(0, 0, 0, 0.14),
    0 1px 5px 0 rgba(0, 0, 0, 0.12);
} */
```

9. You can now reorder the list by dragging and dropping the items.

> For additonal information see: Drag & Drop Documentation

CDK: Virtual Scrolling

(using the ScrollingModule)

The `<cdk-virtual-scroll-viewport>` displays large lists of elements performantly by only rendering the items that fit on-screen. Loading hundreds of elements can be slow in any browser; virtual scrolling enables a performant way to simulate all items being rendered by making the height of the container element the same as the height of total number of elements to be rendered, and then only rendering the items in view. Virtual scrolling is different from strategies like infinite scroll where it renders a set amount of elements and then when you hit the end renders the rest.

Steps

1. Add the following code:

```
//app.component.ts
import { Component } from "@angular/core";
import * as faker from "faker";

@Component({
  selector: "app-root",
  template: `
    <cdk-virtual-scroll-viewport itemSize="50" class="list">
      <div *cdkVirtualFor="let item of items" class="list-item">
        {{ item }}
      </div>
    </cdk-virtual-scroll-viewport>
  `,
  styles: [
    `
      .list {
        height: 200px;
        width: 200px;
        border: 1px solid black;
      }

      .list-item {
        height: 50px;
      }
    `,
  ],
})
export class AppComponent {
```

```
  items = Array.from({ length: 100000 }).map(() =>
`${faker.name.findName()}`);
}
```

> The key to the above example is that the `itemSize` property on the `<cdk-virtual-scroll-viewport>` must match the `.list-item` css class height.

4. View in the Chrome browser and open Chrome DevTools to the Elements tab
5. Scroll through the list and watch the items being generated as they come into view

> For additonal information see: Scrolling Module Documentation

Resources

- Live drag/drop demo video
- Article on Angular Material Setup
- Slides: The CDK is the Coolest Thing You are not Using
- Infinite Virtual Scroll using Angular CDK

Bootstrapping an Application

```
import "./polyfills";

// app/app.component.ts
import { Component } from "@angular/core";

@Component({
  selector: "app-root",
  template: "<h1>Hello Angular</h1>",
})
export class AppComponent {}

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

// app/app.module.ts
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  bootstrap: [AppComponent],
})
export class AppModule {}

// app/main.ts
import { platformBrowserDynamic } from "@angular/platform-browser-dynamic";
platformBrowserDynamic().bootstrapModule(AppModule);
```

Lifecycle Hooks

A component has a lifecycle managed by Angular.

Angular **creates** and renders components along with their children, checks when their data-bound *properties* **change**, and **destroys** them before removing them from the DOM.

Angular offers **lifecycle hooks** that provide visibility into these key life moments and the ability to act when they occur.

Starting Point

Open `demos\start\` directory.

```
git checkout input-property -f
git clean -df
```

Init & Changes

src\app\app.component.ts

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

@Component({
  selector: "app-root",
  template: `
    <app-fruit-list [fruits]="data"></app-fruit-list>
    <button (click)="onClickChange()">Change List</button>
  `,
  styles: [],
})
export class AppComponent {
  data: string[] = ["Apple", "Orange", "Plum"];
  onClickChange() {
    this.data = ["Banana", "Kiwi", "Grape"];
  }
}
```

src\app\fruit-list\fruit-list.component.ts

```
import {
  Component,
  OnInit,
  Input,
  OnChanges,
  SimpleChanges,
} from "@angular/core";

@Component({
  selector: "app-fruit-list",
  template: `
    <ul>
      <li *ngFor="let fruit of fruits">
        {{ fruit }}
      </li>
    </ul>
  `,
  styles: [],
})
export class FruitListComponent implements OnInit, OnChanges {
  @Input()
  fruits: string[];
  constructor() {
    console.log("Constructor");
  }
  ngOnChanges(changes: SimpleChanges): void {
    console.log("OnChanges");
    console.log("Previous Values: " + changes.fruits.previousValue);
    console.log("Current Values: " + changes.fruits.currentValue);
  }

  ngOnInit() {
    console.log("OnInit");
  }
}
```

Results

Refresh Page:

```
Constructor
OnChanges
Previous Values: undefined
Current Values: ["Apple", "Orange", "Plum"]
OnInit
```

Click Change List:

```
OnChanges
Previous Values: ["Apple", "Orange", "Plum"]
Current Values: ["Banana", "Kiwi", "Grape"]
```

Destroy

src\app\app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <app-fruit-list [fruits]="data"
+   *ngIf="showList"></app-fruit-list>
    <button (click)="onClickChange()">Change List</button>
+    <br />
+    <button (click)="onClickRemove()">Remove</button>
  `,
  styles: []
})
export class AppComponent {
  data: string[] = ['Apple', 'Orange', 'Plum'];
  showList = true;
  onClickChange() {
    this.data = ['Banana', 'Kiwi', 'Grape'];
  }
+  onClickRemove() {
+    this.showList = !this.showList;
+  }
}
```

src\app\fruit-list\fruit-list.component.ts

```
import {
  Component,
  OnInit,
  Input,
  OnChanges,
```

```
  SimpleChanges,
+  OnDestroy
} from '@angular/core';

@Component({
  selector: 'app-fruit-list',
  template: `
    <ul>
      <li *ngFor="let fruit of fruits">
        {{ fruit }}
      </li>
    </ul>
  `,
  styles: []
})
export class FruitListComponent implements OnInit, OnChanges,
+ OnDestroy {
  @Input()
  fruits: string[];
  constructor() {
    console.log('Constructor');
  }
  ngOnChanges(changes: SimpleChanges): void {
    console.log('OnChanges');
    console.log('Previous Values: ' + changes.fruits.previousValue);
    console.log('Current Values: ' + changes.fruits.currentValue);
  }
  ngOnInit() {
    console.log('OnInit');
  }
+  ngOnDestroy(): void {
+    console.log('Destroyed');
+  }
}
```

Results

Click Remove

```
Destroyed
```

Final Code

src\app\app.component.ts

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

@Component({
  selector: "app-root",
  template: `
    <app-fruit-list [fruits]="data" *ngIf="showList"></app-fruit-list>
    <button (click)="onClickChange()">Change List</button>
    <br />
    <button (click)="onClickRemove()">Remove</button>
  `,
  styles: [],
})
export class AppComponent {
  data: string[] = ["Apple", "Orange", "Plum"];
  showList = true;
  onClickChange() {
    this.data = ["Banana", "Kiwi", "Grape"];
  }
  onClickRemove() {
    this.showList = !this.showList;
  }
}
```

src\app\fruit-list\fruit-list.component.ts

```
import {
  Component,
  OnInit,
  Input,
  OnChanges,
  SimpleChanges,
  OnDestroy,
} from "@angular/core";

@Component({
  selector: "app-fruit-list",
  template: `
    <ul>
      <li *ngFor="let fruit of fruits">
        {{ fruit }}
      </li>
    </ul>
  `,
  styles: [],
})
export class FruitListComponent implements OnInit, OnChanges, OnDestroy {
```

```
  @Input()
  fruits: string[];
  constructor() {
    console.log("Constructor");
  }
  ngOnChanges(changes: SimpleChanges): void {
    console.log("OnChanges");
    console.log("Previous Values: " + changes.fruits.previousValue);
    console.log("Current Values: " + changes.fruits.currentValue);
  }
  ngOnInit() {
    console.log("OnInit");
  }
  ngOnDestroy(): void {
    console.log("Destroyed");
  }
}
```

ViewChild(ren) & AfterViewInit

```
ng g c hello-world
```

src\app\app.component.ts

```
import { Component, ViewChild, OnInit, AfterViewInit } from "@angular/core";
import { FruitListComponent } from "./fruit-list/fruit-list.component";
import { HelloWorldComponent } from "./hello-world/hello-world.component";

@Component({
  selector: "app-root",
  template: `
    <app-hello-world></app-hello-world>
    <app-fruit-list [fruits]="data" *ngIf="showList"></app-fruit-list>
    <button (click)="onClickChange()">Change List</button>
    <br />
    <button (click)="onClickRemove()">Remove</button>
  `,
  styles: [],
})
export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild(HelloWorldComponent, { static: true }) helloWorldComponent;
  @ViewChild(FruitListComponent, { static: false }) fruitListComponent;
  data: string[] = ["Apple", "Orange", "Plum"];
  showList = true;
```

```
  ngOnInit(): void {
    console.log("App OnInit: ");
    console.log("ViewChild (hello):", this.helloWorldComponent);
    console.log("ViewChild: (fruit list)", this.fruitListComponent);
  }

  ngAfterViewInit(): void {
    console.log("App AfterViewInit: ");
    console.log("ViewChild (hello):", this.helloWorldComponent);
    console.log("ViewChild: (fruit list)", this.fruitListComponent);
  }

  onClickChange() {
    this.data = ["Banana", "Kiwi", "Grape"];
  }
  onClickRemove() {
    this.showList = !this.showList;
  }
}
```

src\app\fruit-list\fruit-list.component.ts

```
import {
  Component,
  OnInit,
  Input,
  OnChanges,
  SimpleChanges,
  OnDestroy,
} from "@angular/core";

@Component({
  selector: "app-fruit-list",
  template: `
    <ul>
      <li *ngFor="let fruit of fruits">
        {{ fruit }}
      </li>
    </ul>
  `,
  styles: [],
})
export class FruitListComponent implements OnInit, OnChanges, OnDestroy {
  @Input()
  fruits: string[];
  constructor() {
```

```
    console.log("Constructor");
  }
  ngOnChanges(changes: SimpleChanges): void {
    console.log("FruitList OnChanges");
    console.log("Previous Values: " + changes.fruits.previousValue);
    console.log("Current Values: " + changes.fruits.currentValue);
  }
  ngOnInit() {
    console.log("FruitList OnInit");
  }
  ngOnDestroy(): void {
    console.log("FruitList Destroyed");
  }
}
```

ContentChild(ren) & AfterContentInit

- ViewChildren don't include elements that exist within the ng-content tag.
- ContentChildren includes only elements that exists within the ng-content tag.

src\hello-world\hello-world.component.ts

```
import {
  Component,
  OnInit,
  ContentChild,
  AfterViewInit,
  AfterContentInit,
} from "@angular/core";

@Component({
  selector: "app-hello-world",
  template: ` <p>Hello World! My name is: <ng-content></ng-content></p> `,
  styles: [],
})
export class HelloWorldComponent
  implements OnInit, AfterViewInit, AfterContentInit {
  @ContentChild("nameContent", { static: true }) nameContent;
  constructor() {}

  ngOnInit() {
    console.log(
      "OnInit: nameContent available only if static is true. ",
      this.nameContent
    );
```

```
  }
  ngAfterContentInit() {
    console.log("AfterContentInit: nameContent available. ",
this.nameContent);
  }
  ngAfterViewInit() {
    console.log("AfterViewInit: nameContent available. ", this.nameContent);
  }
}
```

src\app\fruit-list\fruit-list.component.ts

```
import {
  Component,
  OnInit,
  Input,
  OnChanges,
  SimpleChanges,
  OnDestroy,
} from "@angular/core";

@Component({
  selector: "app-fruit-list",
  template: `
    <ul>
      <li *ngFor="let fruit of fruits">
        {{ fruit }}
      </li>
    </ul>
  `,
  styles: [],
})
export class FruitListComponent implements OnInit, OnChanges, OnDestroy {
  @Input()
  fruits: string[];
  constructor() {
    console.log("Constructor");
  }
  ngOnChanges(changes: SimpleChanges): void {
    console.log("FruitList OnChanges");
    console.log("Previous Values: " + changes.fruits.previousValue);
    console.log("Current Values: " + changes.fruits.currentValue);
  }
  ngOnInit() {
    console.log("FruitList OnInit");
  }
  ngOnDestroy(): void {
```

```
    console.log("FruitList Destroyed");
  }
}
```

src\app\app.component.ts

```
import { Component, ViewChild, OnInit, AfterViewInit } from "@angular/core";
import { FruitListComponent } from "./fruit-list/fruit-list.component";
import { HelloWorldComponent } from "./hello-world/hello-world.component";

@Component({
  selector: "app-root",
  template: `
    <app-hello-world>
      <h2 #nameContent>Bond, James Bond</h2>
    </app-hello-world>
  `,
  styles: [],
})
export class AppComponent implements OnInit, AfterViewInit {
  @ViewChild(HelloWorldComponent, { static: true }) helloWorldComponent;

  ngOnInit(): void {
    console.log("App OnInit: ");
    console.log("ViewChild (hello):", this.helloWorldComponent);
  }

  ngAfterViewInit(): void {
    console.log("App AfterViewInit: ");
    console.log("ViewChild (hello):", this.helloWorldComponent);
  }
}
```

Reference

- Lifecycle Hooks Documentation
- Understanding View Children and ContentChildren

Libraries

Your First Library

1. Run the commands:

```
ng new my-workspace --createApplication="false"
cd my-workspace
ng generate application my-first-app
ng generate library my-lib
ng build my-lib
```

2. Import `MyLibModule` in the app.

my-first-app\src\app\app.module.ts

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

import { AppComponent } from './app.component';
+ import { MyLibModule } from 'my-lib';

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

3. Delete the html in the AppComponent template and add the component from `my-lib`.

my-first-app\src\app\app.component.html

```
<lib-my-lib></lib-my-lib>
```

4. Change the lib component.
5. Notice the app doesn't update.
6. Rebuild and the app updates.

```
ng build my-lib
```

> No need to stop and start `ng serve`

OR

```
ng build my-lib --watch
```

> Library triggers rebuild automatically when changed.

Issues

If you receive this error after running the command `ng build my-lib`:

```
BrowserslistError: Unknown version 67 of android
```

- Delete `node_modules` and `package-lock.json`
- Ensure these versions:

```
"devDependencies": {
"@angular-devkit/build-angular": "^0.801.3",
"@angular-devkit/build-ng-packagr": "^0.801.3",
...
```

- Run `npm install` again.
- See this github issue for more information

Reference

- File Structure: Library Project Files

- Creating Libraries Official Documentation
- Using npm Link
- Creating Your Own Libraries with the Angular CLI

IVY

Ivy is the code name for Angular's next-generation compilation and rendering pipeline. Starting with Angular version 8, you can choose to opt in to start using a preview version of Ivy and help in its continuing development and tuning.

It is the default rendering engine in an Angular version 9 project.

Using Ivy in an existing project

To update an existing project:

1. Update to Angular version 11 using the directions here: https://update.angular.io/

   > Choose From 8.0 to 9.x Then choose from 9.x to 10.x Then choose from 10.x to 11.x Note: You cannot skip versions when upgrading

OR

2. Set the `enableIvy` option in the `angularCompilerOptions` in your project's `tsconfig.app.json`.

```
{
  "compilerOptions": { ... },
  "angularCompilerOptions": {
    "enableIvy": true
  }
}
```

AOT compilation with Ivy is faster and should be used by default. In the `angular.json` workspace configuration file, set the default build options for your project to always use AOT compilation.

```
{
  "projects": {
    "my-existing-project": {
      "architect": {
        "build": {
          "options": {
            ...
```

```
            "aot": true,
          }
        }
      }
    }
  }
}
```

To stop using the Ivy compiler, set `enableIvy` to `false` in `tsconfig.app.json`, or remove it completely. Also remove `"aot": true` from your default build options if you didn't have it there before.

Resources

- A Plan for Version 8 & Ivy
- Ivy Official Documentation
- Ivy Architecture
- The New Ivy Compiler Finally Works on Windows
- What is Angular Ivy
- Exploring the new Ivy Compiler (slightly outdated)

Setup

Creating this project

Not done in class just checkout start branch.

```
ng new demos --routing --inline-style --inline-template --skip-tests --skip-install
```

> Notice the flags --inline-style --inline-template so separate html and css files are not generated for each component.

1. Open `package.json` and remove `devDependencies` related to testing.
2. Delete e2e folder.
3. npm install

Creating the http-start branch

Not done in class just checkout http-start branch.

```
git checkout start
npm install json-server
mkdir api
```

Copy db.json from here: https://jsonplaceholder.typicode.com/db into the api folder.

Add script to `package.json`

```
"scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e",
+   "api": "json-server ./api/db.json"
  },
```

Feedback

I appreciate all feedback. All pdf manuals are protected but are able to be annotated. Send feedback to craig@funnyant.com