Angular Dependencies

By: Mosh Hamedani

In **package.json,** you find these dependencies in our Angular app:

systemjs

This is the module loader we use in our application. With a module loader, we no longer have to add tens or hundreds of script tags (in the right order) in our index.html. Instead, we import the main or starting module of our application into a module loader (like system.js), and based on the “import” statements we use in our code (to bring other modules), it’ll automatically downloading those modules in the right order. In production applications, we can also have our module loader create a bundle of all our application modules. There are many module loaders out there, and a few populate ones include: **browserify**, **webpack**, and **system.js**. System.js is the one that Angular team prefers.

es6-promise

The next version of Javascript (es6) introduces a nice and easy API to work with promises. We use promises to handle the results and errors of asynchronous operations (eg AJAX calls, web workers, etc). This library (es6-promise) is a polyfill for es6 promises, meaning it brings es6 promises into es5.

es6-shim

Apart from promises, es6 brings many other useful features, like modules, classes, etc. This library provides compatibility shims so current JS engines behave as close as possible to es6.

Angular Dependencies

By: Mosh Hamedani

reflect-metadata

With es7, we will be able to add metadata (also called annotation or decorator) to a class or function. This library is a polyfill for metadata API in es7.

```
@Component({ selector: “app” })
class AppComponent { }
```

rxjs

This is Reactive Extensions for Javascript library, which introduces an elegant way to work with asynchronous operations. We have a complete section on Reactive Extensions and observables later in the course. Angular uses a concept called “observables” in the implementation of its Http and Jsonp classes. Angular 1, we used promises, but Angular 2 embraces observables. You’ll find out why later in the course.

zone.js

One of the complexities of Angular 1 is its digest loop, which helps Angular 1 figure out the changes in the model and refresh the view. In Angular 2 we don’t have this concept anymore. In Angular 2, all browser events are “monkey-patched”, so even if you an event handler outside your Angular app, it will still be notified and can detect changes in objects. Angular team have extracted this part from the core Angular script and open-sourced it as a separate library that can be re-used by others. This library is called zone.js.

Angular Basics

By: Mosh Hamedani

Creating Components

```
import {Component} from ‘angular2/core’

@Component({
     selector: ‘courses’,
     template: ‘<h2>Courses</h2>
})
export class Component { }
```

Using Components

In AppComponent:

```
import {CoursesComponent} from ‘./courses.component’

@Component({
      template: ‘<courses></courses>’,
      directives: [CoursesComponent]
})
```

Angular Basics

By: Mosh Hamedani

Templates

Interpolation

```
{{ title }}
```

Displaying lists:

```
<ul>
     <li *ngFor=“#course of courses”>
         {{ course }}
     </li>
</ul>
```

Services

```
export class CourseService {

}
```

Dependency Injection

```
@Component({
     providers: [CourseService]
})
export class CourseComponent {
      constructor(courseService: CourseService) {
      }
}
```

Directives

Basic structure

```
import {Directive} from ‘angular2/core’

@Directive({
    selector: [autoGrow],
    host: {
        ‘(focus)’: ‘onFocus()’,
        ‘(blur)’: ‘onBlur()’
    }
})
export class AutoGrowDirective {
     onFocus() {
     }

     onBlur() {
     }
}
```

To access and modify DOM elements

```
import {ElementRef, Renderer} from ‘angular2/core’


export class AutoGrowDirective {
     constructor(el: ElementRef, renderer: Renderer) {
     }

     onFocus(){
        this.renderer.setElementStyle(this.el, ‘width’,
‘200’);
     }
}
```

Angular Bindings

By: Mosh Hamedani

Interpolation

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

Property binding

```
<img [src]=“imageUrl” />
<img bind-src=“imageUrl” />
```

Class binding

```
<li [class.active]=“isActive” />
```

Style binding

```
<button [style.backgroundColor]=“isActive ? ‘blue’ : ‘gray’”>
```

Event binding

```
<button (click)=“onClick($event)”>
<button on-click=“onClick($event)”>
```

Two-way binding

```
<input type=“text” [(ngModel)]=“firstName”>
<input type=“text” bindon-ngModel=“firstName”>
```

Components

By: Mosh Hamedani

Input Properties

Using @Input annotation

```
import {Input} from ‘angular2/core’;

@Component(…)
export class FavoriteComponent {
    @Input(‘is-favorite’) isFavorite = false;
}
```

Using component metadata

```
@Component({
    inputs: [‘isFavorite:is-favorite’]
})
export class FavoriteComponent {
    isFavorite = false;
}
```

In the host component

```
<favorite [is-favorite]=“post.isFavorite”></favorite>
```

Components

By: Mosh Hamedani

Output Properties

Using @Output annotation

```
import {Output} from ‘angular2/core’;

@Component(…)
export class FavoriteComponent {
    @Output(‘favorite-change’) change = new EventEmitter();

    onClick() {
        this.change.emit({ newValue: this.isFavorite });
    }
}
```

Using component metadata

```
@Component({
    outputs: [‘change:favoriteChange’]
})
export class FavoriteComponent {
    change = new EventEmitter();

    onClick() {
        this.change.emit({ newValue: this.isFavorite });
    }
}
```

Components

By: Mosh Hamedani

In the host component

```
<favorite (favoriteChange)=“onChange()”></favorite>
```

Templates

```
@Component({
      template: ‘…’, // or
      templateUrl: ‘app/template.template.html’
})
```

Styles

```
@Component({
      styles: [‘…’],
      styleUrls: [‘…’, ‘…’];
})
```

Templates

By: Mosh Hamedani

Showing / hiding elements

```
<div [hidden]="courses.length == 0"></div>

<div *ngIf="courses.length > 0"></div>


<div [ngSwitch]="viewMode">
     <template [ngSwitchWhen]="'map'" ngSwitchDefault>
        …
     </template>
     <template [ngSwitchWhen]="'list'">
        …
     </template>
</div>
```

Pipes

```
{{ course.title | uppercase }} -> ANGULAR COURSE
{{ course.students | number }} -> 1,234
{{ course.rating | number:'2.2-2' }} -> 04.97
{{ course.price | currency:'USD':true }} -> $99.95
{{ course.releaseDate | date:'MMM yyyy' }} -> Mar 2016
{{ course | json }}
```

Templates

By: Mosh Hamedani

Creating Custom Pipes

```
import {Pipe, PipeTransform} from ‘angular2/core’;


@Pipe({ name: ‘summary’ })
export class SummaryPipe implements PipeTransform {
     transform(value: string, args: string[]) {
     }
}
```

In the host component

```
@Component({
    pipes: [SummaryPipe]
})
```

Applying multiple classes dynamically

```
<i [ngClass]=“{
    active: isActive,
    disabled: isDisabled
}”></i>
```

Templates

By: Mosh Hamedani

Applying multiple styles dynamically

```
<button
    [ngStyle]="{
        backgroundColor: canSave ? 'blue': 'gray',
        color: canSave ? 'white' : 'black'
}"></button>
```

Elvis operator

```
{{ task.assignee?.role?.name }}
```

Inserting content from the outside

```
<ng-content></ng-content>
```

Multiple content placeholders

```
<ng-content select=".heading"></ng-content>
<ng-content select=".body"></ng-content>
```

Model-driven Form Exercise

By: Mosh Hamedani

Implementing a custom validator

In this section, you learned how to create a custom validator. We define a method that takes a **Control** object:

```
static cannotContainSpace(control: Control){
     if (valid)
        return null;

     return { cannotContainSpace: true };
}
```

Then, we reference this method when creating a **Control** using **FormBuidler**:

```
this.form = fb.group({
     username: [‘’, UsernameValidators.cannotContainSpace]
});
```

If we need multiple validators, we compose them:

```
this.form = fb.group({
     username: [‘’, Validators.compose([
         Validators.required,
         UsernameValidators.cannotContainSpace
     ])
});
```

Model-driven Form Exercise

By: Mosh Hamedani

Comparing fields

The validator method we've implemented here takes a **Control** object, which represents a single field in the form. To compare two fields, you need to implement a validation method that takes a **ControlGroup**. Then, we can access any **Controls** in that group:

```
static myCustomValidator(group: ControlGroup){
     var control1 = group.find('control1');
     var control2 = group.find('control2');
     …
}
```

We can then pass this validator, when creating a group using **FormBuilder**:

```
this.form = fb.group({
…
}, { validator: myCustomValidator });
```

So, as the second argument to the **group()** method, we pass an object that contains "**validator**" property. We reference our custom validator method here. This method, like other validator methods can be in the component, or in a separate class (as a static method) for re-usability.

Note: If the form is not functioning properly when doing this exercise, make sure to have a look at the console in developer tools. You might have an error and be totally unaware of it!

Adventures in Rx

By: Mosh Hamedani

In this guide, we'll explore other useful ways to use observables. By the end of this tutorial, you'll learn how to

- Create an observable from DOM events (the Angular way)
- Create an observable from an array
- Create an observable from a list of arguments
- Implement timers
- Run parallel operations
- Handle errors
- Get notified when an observable completes

As part of reading this tutorial, **you should type all the code snippets, run them and see the results.** This will prepare you for the upcoming quiz in this section and the coding exercise in the next section.

Do NOT copy/paste code from this document. Write all the code by hand.

Adventures in Rx

By: Mosh Hamedani

Creating an observable from DOM events

Earlier, you learned how to create an observable from DOM events:

```
var element = $(“#search”);
var observable = Observable.fromEvent(element, “keyup”);
```

This code is tightly coupled to the DOM and is hard to unit test. Client-side code coupled to the DOM is as bad as server-side code that talks directly to a database without a set of abstractions.

In Angular apps, we should never use libraries like jQuery or work directly with the **document** object! The whole point of Angular components is to decouple our code from the DOM. So, we use property binding and let Angular work with the DOM instead.

The reason I used jQuery in the videos was to strip away the unnecessary complexity, so you could focus on the observables and their benefits.

The Angular way

In the section about forms, we looked at the **Control** class: it represents an input field in a form. This class has a property called **valueChanges**, which returns an observable. We can subscribe to this observable and get notified as the value in the input field changes. This way, we will not be working directly with a DOM element.

1- Use the following template in **AppComponent**

```
<form [ngFormModel]="form">
    <input type="text" ngControl="search">
</form>
```

2- Create the **form** object in **AppComponent**:

```
import {ControlGroup, FormBuilder} from 'angular2/common';


form: ControlGroup;


constructor(fb: FormBuilder) {
     this.form = fb.group({
          search: []
     });
}
```

3- Get a reference to the “search” control and subscribe to its **valueChanges** property (in the constructor):

```
var search = this.form.find('search');
search.valueChanges
      .subscribe(x => console.log(x));
```

4- Run the application and type something into the input field. Note that with every keystroke, **valueChanges** observable pushes the current value of the input field.

The **valueChanges** property is also available in the **ControlGroup** class. We can use this to be notified when anything changes in a control group (or in the entire form).

5- Apply the following operators before subscribing to valueChanges. What will you see in the console?

```
     .debounceTime(400)
     .map(str => (<string>str).replace(' ', '-'))
```

Note that **<string>str** casts **str** (which is of type "any") to a string.

Creating an observable from an array

Creating observables is not limited to asynchronous data streams. We can create an observable from an array:

```
var observable = Observable.fromArray([1, 2, 3]);
```

This line will return an observable with three items: 1, 2 and 3. So, when subscribing to this observable, we'll get three values pushed at us.

What is the benefit of converting an array to an observable stream? We can tie this observable stream with another observable created off DOM events, AJAX calls, web sockets, etc. For example, think of a travel search engine like Kayak. Many travel search engines let the user search for flights for exact dates or within a 2-day window.

1- Let's say the user selects the 2-day window. Create an array of the travel start dates with the 2-day window:

```
var startDates = [];
var startDate = new Date(); // Assuming today for simplicity


for (var day = -2; day <= 2; day++) {
    var date = new Date(
        startDate.getFullYear(),
        startDate.getMonth(),
        startDate.getDate() + day);
        startDates.push(date);
}
```

2- Convert this array to an observable, and for each data element, use the map operator to call the server and get the deals for the given date:

```
Observable
    .fromArray(startDates)
    .map(date => console.log("Getting deals for date " + date))
    .subscribe(x => console.log(x));
```

What do you see in the console? Why do we get “undefined” after each call to the server? Because our map operator is not returning any values to the subscribers.

3- Modify the map operator as follows:

```
    .map(date => {
           console.log("Getting deals for date " + date);
           return [1, 2, 3];
        })
```

Here, [1, 2, 3] simulates the flight deals returned from the server for the given date.

Note: For simplicity, I excluded the end date as part of getting the deals.

Other ways to create an observable

We can use the static **Observable.of()** method to create an observable from a list of arguments:

```
// Returns an observable with one item: 1
Observable.of(1);

// Returns an observable with three items: 1, 2, 3
Observable.of(1, 2, 3);

// Returns an observable with one item: [1, 2, 3]
Observable.of([1, 2, 3]);
```

Given the following code snippet:

```
var observable = …
observable.subscribe(x => console.log(x));
```

Explore different ways to create an observable. Make sure you understand how each method works:

```
Observable.empty()
Observable.range(1, 5)
Observable.fromArray([1, 2, 3])
Observable.of([1, 2, 3])
```

Implementing a timer

We can use the static **Observable.interval()** method to create a timer. This is useful for running an asynchronous operation at specified intervals.

1- Create an observable using the interval method:

```
var observable = Observable.interval(1000);
observable.subscribe(x => console.log(x));
```

You'll see a number in the console every 1000 milliseconds. This number is just a zero-based index and doesn't have much value on its own. But we can extend this example and build a client that calls the server once a minute (rather than every second) to get the latest tweets, emails, news feed, etc.

2- Apply the map operator and see the results in the console:

```
observable
    .map(x => {
        console.log("calling the server to get the latest news");
    })
    .subscribe(news => console.log(news));
```

Note that I've also changed the name of the argument of the arrow function in the subscribe method from **x** to **news**.

So, you'll see a message in the console simulating a call to the server every 1000 milliseconds. This is similar to using setInterval() method of Javascript.

Note that our **map** operator doesn't return any values and that's why we get "undefined" in the console.

3- Modify the map operator as follows:

```
.map(x => {
    console.log("calling the server to get the latest news");
    return [1, 2, 3];
})
```

Here we are assuming that [1, 2, 3] are the latest news/tweets/messages since the last time we polled the server.

4- As you've learned, we can create an observable for AJAX calls. Let's simulate an AJAX call by modifying the **map** operator to return an observable from our array. This means we should also replace **map** with **flatMap**, otherwise, we'll end up with an Observable<Observable<number[]>> (observable of observable of number[]).

```
.flatMap(x => {
    console.log("calling the server to get the latest news");
    return observable.of([1, 2, 3]);
})
```

Running Parallel Operations

Let's say you want to build a page and display the user's profile data and their tweets. Assume that your RESTful API, by design, does not return the user's profile and their tweets in a combined format. So you need to send two requests in parallel: one to get the profile, and another to get the tweets.

We can use the **forkJoin** operator to run all observables in parallel and collect their last elements.

```
Observable.forkJoin(obs1, obs2, …)
```

1- Create two observables, one for the user, one for their tweets.

```
var userStream = Observable.of({
      userId: 1, username: 'mosh'
}).delay(2000);
```

```
var tweetsStream = Observable.of([1, 2, 3]).delay(1500);
```

I've used the **delay** operator here to simulate an AJAX call that takes 1500/2000 milliseconds.

2- Use **forkJoin** to run both these observables in parallel. Note the result in the console.

```
Observable
    .forkJoin(userStream, tweetsStream)
    .subscribe(result => console.log(result));
```

3- So, **forkJoin** returns an array of data elements collected from multiple observables. Use the **map** operator to map this array into a data structure that our application expects:

```
Observable
    .forkJoin(userStream, tweetsStream)
    .map(joined =>
             new Object({user: joined[0], tweets: joined[1] }))
    .subscribe(result => console.log(result));
```

Handling Errors

Asynchronous operations that involve accessing the network can fail due to request timeouts, server failures, etc. We can handle such failures by providing a callback function as the second argument to the **subscribe** method:

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

1- Simulate a failed AJAX call, by creating an observable using the static **Observable.throw()** method. This method returns an observable that terminates with an exception.

```
var observable = Observable.throw(new Error("Something 
failed."));
```

2- Subscribe to this observable and provide an error handler:

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

Note the result in the console.

Retrying

We can retry a failed observable using the **retry** operator.

```
observable.retry(3)


// Retry indefinitely
observable.retry()
```

1- Simulate an AJAX call that fails twice and succeeds the third time.

```
var counter = 0;


var ajaxCall = Observable.of('url')
    .flatMap(() => {
         if (++counter < 2)
            return Observable.throw(new Error("Request failed"));


         return Observable.of([1, 2, 3]);
    });
```

So, here, **counter** is just a local variable we're using for simulating an AJAX call that fails the first two times. The third time it returns the array [1, 2, 3].

In a real-world application, you wouldn't create an observable for an AJAX call like this. In Angular, we have a class called **Http,** which we use for making AJAX calls. You'll learn about this class in the next section. All methods of the **Http** class return an observable.

2- Subscribe to this observable and note the result in the console:

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

3- Use the **retry** operator before subscribing and note the difference:

```
ajaxCall
    .retry(3)
    .subscribe(
         x => console.log(x),
         error => console.error(error));
```

Imagine if you had to implement the retry imperatively (using logic). At a minimum, you would need a loop and a conditional statement. With observables, it's just a matter of calling the **retry()** method. So, observables makes dealing with asynchronous operators easier.

Catching and continuing

We can use the **catch** operator to continue an observable that is terminated by an exception with the next observable.

For example, in a real-world application, we may call a remote service to get some data, but if the call fails for any reason, we may want to get the data from local storage or present some default data to the user. We can create another observable from an array and use the **catch** operator to continue the first failed observable with the second observable.

1- Run the following code snippet and see the result:

```
var remoteDataStream = Observable.throw(new Error("Something
failed."));


remoteDataStream
    .catch(err => {
        var localDataStream = Observable.of([1, 2, 3]);
        return localDataStream;
    })
    .subscribe(x => console.log(x));
```

So, if **remoteDataStream** throws an exception, we'll catch it and then return another observable (**localDataStream**).

2- Modify the creation of **remoteDataStream** as follows:

```
var remoteDataStream = Observable.of([4, 5, 6]);
```

Do you see the difference? So, the **catch** operator is only called if the first observable fails.

Timeouts

What if we call a remote service that doesn't always respond in a timely fashion? We don't want to keep the user waiting. We can use the **timeout** operator:

```
// times out after 200 milliseconds.
observable.timeout(200);
```

1- Simulate a long running operation:

```
var remoteDataStream = Observable.of([1, 2, 3]).delay(5000);
```

2- Use the **timeout** operator and then subscribe to this observable:

```
remoteDataStream
    .timeout(1000)
    .subscribe(
        x => console.log(x),
        error => console.error(error)
    );
```

With this, the observable times out after 1 second and you'll get the error message in the console.

Getting notified when an observable completes

Observables can go in the "completed" state, which means they will no longer push data items in the future. An observable wrapping an AJAX call is an example of an observable that completes. If an AJAX calls succeeds, the response will be placed in the observable and pushed at the subscriber. And if it fails, the observable will terminate with an exception. Either way, the observable completes and it will no longer emit any values or errors.

We can get notified when an observable completes by providing a third argument to the subscribe method:

```
observable.subscribe(
     x => console.log(x),
     error => console.log(error),
     () => console.log(“completed”));
```

Use this **subscribe** method with the following observables and note the messages logged in the console:

```
Observable.throw(new Error(“error”))
Observable.fromArray([1, 2, 3])
```

Summary

In this guide, you learned about:

- The **valueChanges** property of **Control** and **ControlGroup** objects. This property is an observable that we can subscribe to, to get notified when the value of an input field changes.

- Various ways to create an observable

```
Observable.fromArray([1, 2, 3])
Observable.of(1)
Observable.of(1, 2, 3)
Observable.of([1, 2, 3])
Observable.empty()
Observable.range(1, 3)
```

- Creating timers using **Observable.interval()**

```
Observable.interval(1000)
    .map(…)
```

- Running parallel operations using **Observable.forkJoin()**

```
Observable.forkJoin(obs1, obs2)
```

- **Observable.throw()** to return an observable that terminates with an exception.

- Handling errors by supplying a callback to the **subscribe()** method

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

- Retrying

```
 obs.retry(3).subscribe()
```

- Catching and continuing

```
obs.catch(error => {
      return anotherObservable;
})
```

- Using timeouts:

```
obs.timeout(1000).subscribe()
```

- Getting notified when an observable completes:

```
obs.subscribe(
    x => console.log(x),
    error => console.error(error),
    () => console.log("completed")
)
```

CORS and JSONP

By: Mosh Hamedani

JSONP

- https://johnnywey.wordpress.com/2012/05/20/jsonp-how-does-it-work/
- http://schier.co/blog/2013/09/30/how-jsonp-works.html

CORS

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS#Access-Control-Request-Headers
- https://remysharp.com/2011/04/21/getting-cors-working

Connecting to the Server

By: Mosh Hamedani

Adding the Http script:

In **index.html**:

```
<script src="node_modules/angular2/bundles/http.dev.js"/>
```

Http Class

```
http.get(url)
     .map(response => response.json());
```

```
http.post(url, JSON.stringify(obj))
     .map(response => response.json());
```

Other methods:

```
http.put()
http.delete()
http.patch()
```

Connecting to the Server

By: Mosh Hamedani

Using Http Class

In the service:

```
import {Http} from ‘angular2/http’;


@Injectable()
export class PostService {
    constructor(private _http: Http) {}

    getPosts() {
        return this._http.get(url).map(res => res.json());
    }
}
```

In the component:

```
import {HTTP_PROVIDERS} from ‘angular2/core’;


@Component({
    providers: [PostService, HTTP_PROVIDERS]
})
export class AppComponent implements OnInit {
     posts: any[];

     ngOnInit() {
         this._postService.getPosts()
              .subscribe(posts => this.posts = posts);
     }
}
```

Connecting to the Server

By: Mosh Hamedani

Component’s Lifecycle Hooks

- **OnInit**: to execute logic after component’s data-bound properties have been initialized.

- **OnChanges**: to execute logic if any bindings have changed

- **AfterContentInit**: to execute logic when a component’s content (<ng-content>) has been fully initialized.

- **AfterContentChecked**: to execute logic after every check of component’s content.

- **AfterViewInit**: to execute logic when a component’s view has been fully initialized.

- **AfterViewChecked**: to execute logic after every check of component’s view.

- **OnDestroy**: to execute clean up logic when a component is destroyed.

These interfaces have a method with the same name as the interface name, prefixed with “ng”.

```
export class AppComponent implements OnInit {
    ngOnInit() {
    }
}
```

Showing a Loader Icon

```
export class AppComponent {
    isLoading = true;

    ngOnInit() {
        this._postService.getPosts()
            .subscribe(posts => {
                this.posts = posts
                this.isLoading = false;
            });
    }
}
```

In the view:

```
<i *ngIf=“isLoading” class=“fa fa-spinner fa-spin fa-3x”></i>
```

Connecting to the Server

By: Mosh Hamedani

Jsonp Class

Only supports get().

In the component:

```
import {JSONP_PROVIDERS} from ‘angular2/core’;

@Component({
    providers: [PostService, JSONP_PROVIDERS]
})
```

Adding Custom Request Headers

```
var headers = new Headers({
    “key”: “value”
});

var options = new RequestOptions({ headers: headers });

http.get(url, options);
```

Exercise Hints

Connecting to the Server

By: Mosh Hamedani

GitHub API endpoints

`Users: https://api.github.com/users/octocat`

`Followers: https://api.github.com/users/octocat/followers`

Note that in both these endpoints, we're using **octocat** username. Design your service such that you can reuse it to get any user's profile and followers.

Importing forkJoin

You need to use:

```
import 'rxjs/add/observable/forkJoin';
```

Note that **forkJoin** is actually a static method, and not an operator.

```
// to import an operator
import 'rxjs/add/operator/opertatorName';

// to import a static Observable method
import 'rxjs/add/observable/methodName';
```

Exercise Hints

By: Mosh Hamedani

Avatars

Avatar images on GitHub are actually bigger than what you saw in the video. I've applied the following CSS class to make them smaller and round:

```
.avatar {
    width: 100;
    height: 100;
    border-radius: 100%;
}
```

List of Followers

To render the list of followers in the format you saw in the video, use **Bootstrap Media Object** component:

http://getbootstrap.com/components/#media

You just need to copy the markup from here and paste it into your component. Then, fill the missing parts with interpolation strings.

Loader

Be sure to add a loader icon until the results are ready.

Angular Routers

By: Mosh Hamedani

Enabling Routing

1- Add **<base>** in index.html:

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

2- Add a reference to the router script in **index.html**:

```
<script src="node_modules/angular2/bundles/router.dev.js"/>
```

3- Add router providers when bootstrapping the application (so Angular can do the required dependency injection behind the scene):

In **boot.ts**

```
import {ROUTER_PROVIDERS} from 'angular2/router'


bootstrap(AppComponent, [ROUTER_PROVIDERS]);
```

Angular Routers

By: Mosh Hamedani

Configuring Routes

```
@RouteConfig([
  { path: '/', name: 'Home', component: HomeComponent },
  { path: '/albums', name: 'Albums', component: AlbumsComponent},
  { path: '/albums/:id', name: 'AlbumDetails, component: … },
  { path: '/*other', name: 'Other', redirectTo: ['Home'] }
])
export class AppComponent {}
```

Using Router Outlet

```
import {ROUTER_DIRECTIVES} from 'angular2/router';

@Component({
    template: `
         <router-outlet></router-outlet>
    `,
    directives: [ROUTER_DIRECTIVES]
})
export class AppComponent {
}
```

Angular Routers

By: Mosh Hamedani

Adding Links

Make sure you’ve added ROUTER_DIRECTIVES in the component metadata.

```
<a [routerLink]=“[‘AlbumDetail’, { id: 1 }]”>View</a>
```

Getting Route Parameters

```
import {RouteParams} from ‘angular2/router’;


export class AppComponent {
    constructor(routeParams: RouteParams) {
         var id = routeParams.get(“id”);
    }
}
```

Angular Routers

By: Mosh Hamedani

Imperative Navigation

```
import {Router} from ‘angular2/router’;


export class AppComponent {
    constructor(private _router: Router) {
    }

    onClick() {
       this._route.navigate(‘[Albums]’);
    }
}
```

Router Lifecycle Hooks

```
import {CanActivate, CanDeactivate} from ‘angular2/router’;


export class ContactFormComponent implements CanDeactivate {
    routerCanDeactivate(next, previous) {
        if (this.form.dirty)
            return confirm(‘Are you sure?’);
    }
}
```

By: Mosh Hamedani

Child Routing Components

```
@RouteConfig([
    {
        path: ‘/events/...’,
        name: ‘Events’,
        component: EventsComponent
    }
])
export class AppComponent {}


@RouteConfig([
    {
        path: ‘/’,
        name: ‘EventsList’,
        component: EventsListComponent
    },
    {
        path: ‘/:id’,
        name: ‘EventDetails’,
        component: EventsDetailsComponent
    }
])
export class EventsComponent {}
```

Cleaning a Form Upon Submit

By: Mosh Hamedani

When we save the new user, we need to mark the form as clean or pristine, so we can navigate away from this page without getting the confirmation box for dirty tracking.

Currently, Angular does not support cleaning the state of the form, but soon we'll have a method on the **ControlGroup** class, called **markAsPristine()**.

So, when you create the new user, you should call:

```
this.form.markAsPristine();
```

You can read more about this here:

https://github.com/angular/angular/pull/6679

Styles for list of posts

By: Mosh Hamedani

```
.posts li { cursor: default; }
.posts li:hover { background: #ecf0f1; }


.list-group-item.active,
.list-group-item.active:hover,
.list-group-item.active:focus {
     background-color: #ecf0f1;
     border-color: #ecf0f1;
     color: #2c3e50;
}
```

In the template:

```
<form [ngFormModel]="signupForm">
    …
    <input ngControl="name">
    …
    <input ngControl="email">
    …
    <div ngControlGroup="billing">
        <input ngControl="cardNumber">
        …
        <input ngControl="expiry">
    </div>
</form>
```

Implementing Custom Validators

```
export class UsernameValidators {
    static cannotContainSpace(control: Control) {
         … // validation logic;

         if (invalid)
            return { cannotContainSpace: true });

         return null;
    }
}
```

When creating the control using FormBuilder:

```
name: [‘’, Validators.compose([
    Validators.required,
    UsernameValidators.cannotContainSpace
])]
```

Async Validator

The same as a custom validator, but we return a Promise.

```
static shouldBeUnique(control: Control){
     return new Promise((resolve, reject) => {
         … // validation logic
         if (invalid)
             resolve({ shouldBeUnique: true });
         else
             resolve(null);
     });
}
```

When building a control using FormBuilder:

```
name: [‘’,Validators.required, UsernameValidators.shouldBeUnique]
```

Note the syntax:

```
[defaultValue, customValidators, asyncValidators]
```

When using more than one custom or async validator, we use the **compose** or **composeAsync** methods:

```
[
    defaultValue,
    Validators.compose([v1, v2]),
    Validators.composeAsync([v3, v4])
];
```

To show a loader when an async validator is in progress:

```
<input ngControl="name">
<div *ngIf="name.control.pending">Checking for uniqueness…</div>
```

Updates after Moving Files

By: Mosh Hamedani

If you restructure your project the way I did, there are three more modifications you need to make apart from updating import statements.

I decided to exclude them from the video, because I thought they would tire some students.

Here are the updates:

posts.component.ts

```
templateUrl: ‘app/posts/posts.component.html’
```

user-form.component.ts

```
templateUrl: ‘app/users/user-form.component.html’
```

users.component.ts

```
templateUrl: ‘app/users/users.component.html’
```