Information Technology Institute

ANGULAR 10

Presented by
Eng./Abanoub Nabil
Teaching assistant at ITI

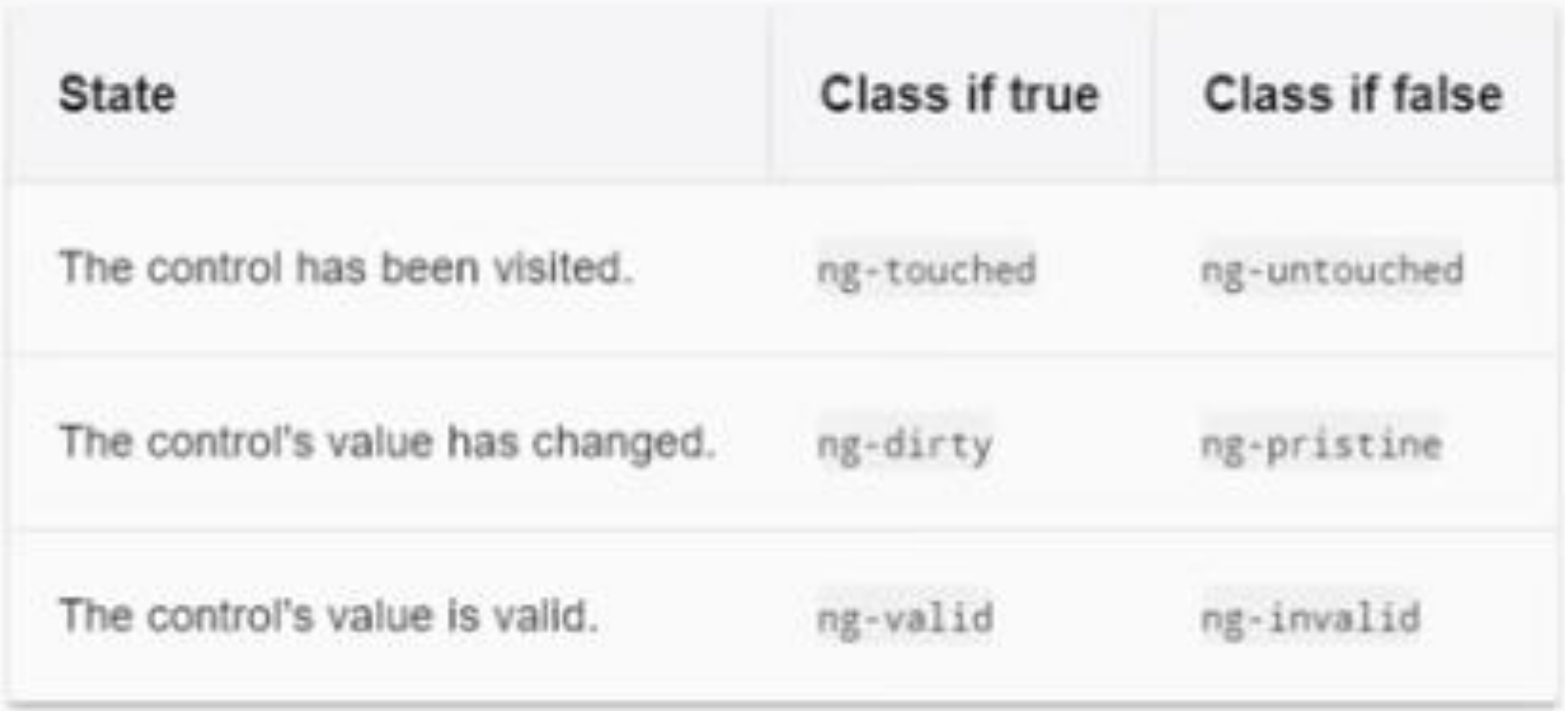
Agenda

- Tracking state and validity
- Providing visual feedback.
- Displaying error messages.
- Posting data to the server.

TDF (Cont.) What we will do?

- **Generate a new CLI project.**
- **Add the form HTML.**
- **Biding data.**
- **Tracking state and validity.**
- Providing visual feedback.
- Displaying error messages.
- Posting data to the server.

4-Tracking state and validity



State	Class if true	Class if false
The control has been visited.	`ng-touched`	`ng-untouched`
The control's value has changed.	`ng-dirty`	`ng-pristine`
The control's value is valid.	`ng-valid`	`ng-invalid`

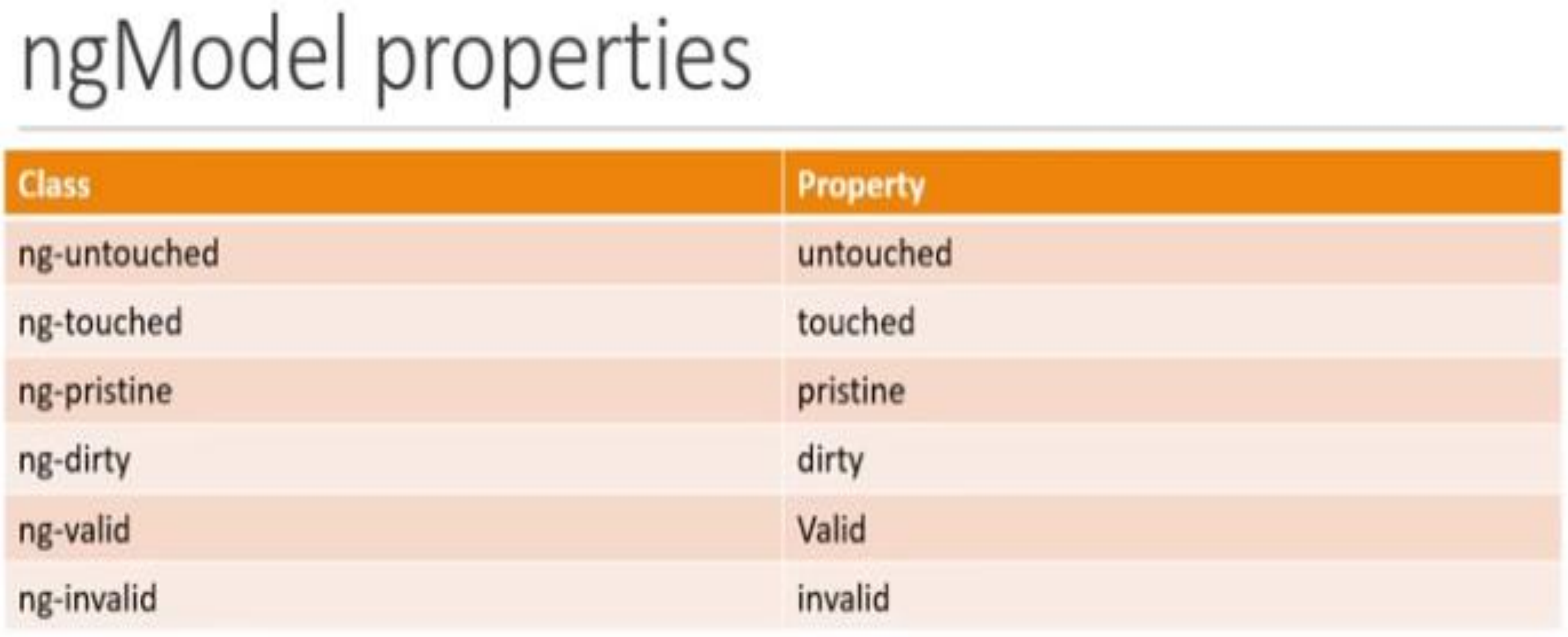
4-Tracking state and validity (cont.)

- If you try to see the default tracking class you can make this by give template reference variable to the input and see className of it.

```
<input type="text" #name  class="form-control"
[(ngModel)]="userModel.name" name="txtName">

{{name.className}}
```

4-Tracking state and validity (cont.)



ngModel properties

Class	Property
ng-untouched	untouched
ng-touched	touched
ng-pristine	pristine
ng-dirty	dirty
ng-valid	Valid
ng-invalid	invalid

4-Tracking state and validity (cont.)

- **How to get access to those ngModel properties?**
- Simply by creating a reference to the ngModel Directive

```
<input type="email" #email="ngModel"
class="form-control"
[(ngModel)]="userModel.email" name="txtEmail">
{{email.untouched}}
```

TDF (Cont.) What we will do?

- **Generate a new CLI project.**
- **Add the form HTML.**
- **Biding data.**
- **Tracking state and validity.**
- **Providing visual feedback.**
- Displaying error messages.
- Posting data to the server.

Providing visual feedback.

- **A good user experiences** is to visually indicate to the user if a form filed is invalid when he enters the details.

- Lets look at some validation of the form field and applying an appreciate class to visually indicate to the user when the form is invalid.

- **We have two approaches**
- You can create your own class with the styles you need or you can use the **validation classes that bootstrap** framework provides.

Providing visual feedback (cont.)

- The first class is **is-invalid** class but we must ensure that validation classes applied to the form control **when it is invalid only** we make that by using **ngModel** properties coupled with class binding.

```
<input type="text" required  [class.is-invalid]="name.invalid"
#name="ngModel" class="form-control"
[(ngModel)]="userModel.name" name="txtName"/>
```

5-Providing visual feedback (cont.).

- We used required in name form control now let us use pattern to support regular expression.
- First of all we must make a reference to ngModel **#phone="ngModel“**

- Then apply the pattern **pattern="^\d{10}$"**
- Then apply is invalid class

  [class.is-invalid]="phone.invalid"

5-Providing visual feedback (cont.).

```
<input type="tel" class="form-control" #phone="ngModel"
pattern="^\d{10}$" [class.is-invalid]="phone.invalid"
[(ngModel)]="userModel.phone" name="txtTel">
```

TDF (Cont.) What we will do?

- **Generate a new CLI project.**
- **Add the form HTML.**
- **Biding data.**
- **Tracking state and validity.**
- **Providing visual feedback.**
- **Displaying error messages.**
- Posting data to the server.

Displaying error messages.

- In this part we will learn how to display error messages when a form field is invalid.
- The name field is required so we need when it is invalid show message to the user

name field is required

Displaying error messages(cont.).

- The first step is to add the message in html after the input element.

  <small>Name is Required</small>

- Now we want this message to be shown when only the field is invalid so we will use conditions of **ngModel** property we can use ngIf or class binding.

```
<small [class.d-none]="name.valid ||
name.untouched">Name is Required</small>
```

Displaying error messages(cont.).

- The last thing add class to the error message to be appeared as error message style.

```
<input type="text" required [class.is-invalid]="name.invalid
&&name.touched" #name="ngModel" class="form-control"
[(ngModel)]="userModel.name" name="txtName"/>



<small class="text-danger" [class.d-none]="name.valid ||
name.untouched">Name is Required</small>
```

Displaying error messages(cont.).

- What if we need to show a specific error for each case to the same form control element?

Displaying error messages(cont.).

```
<div class="form-group">
<label>Telephone :</label>
<input type="tel" [class.is-invalid]="phone.invalid && phone.touched"
class="form-control" #phone="ngModel" pattern="^\d{10}$" required [class.is-
invalid]="phone.invalid" [(ngModel)]="userModel.phone" name="txtPhone">
    <div *ngIf="phone.errors && (phone.invalid ||
    phone.touched)">
        <small class="text-danger"
        *ngIf="phone.errors.required">Phone   number is
        required</small>
        <small class="text-danger"
        *ngIf="phone.errors.pattern">Phone number must be 10
        digits</small>
    </div>
</div>
```

6-Displaying Error Messages(cont.)

- **Select control validation**
- In this part we are going to lean how validate select control.
- The easiest way is to use the **required** attribute in the select tag and then create a reference to ngModel.

<select class="custom-select" **required** **#topic="ngModel"**..........

6-Displaying Error Messages(cont.)

- Then we can use now class binding.

```
<select class="custom-select"
[class.is-invalid]="topic.invalid &&
topic.touched" required    #topic="ngModel"
[(ngModel)]="userModel.topic" name="DDLTopics">
```

6-Displaying Error Messages(cont.)

- Then display the error message.

```
<small class="text-danger" [class.d-
none]="topic.valid || topic.untouched">
Please choose a topic</small>
```

6-Displaying Error Messages(cont.)

- But this will work only with the empty default values.
- `<option value="">I'm Interested In</option>`
- If we add value the validation will fail.
- So what we will do is to handle **blur** and **change** events.

6-Displaying Error Messages(cont.)

- <select **(blur)="ValidateTopic(topic.value)" (change)="ValidateTopic(topic.value)“** class="custom-select" [class.is-invalid]="topic.invalid && topic.touched" required #topic="ngModel" [(ngModel)]="userModel.topic" name="DDLTopics">

6-Displaying Error Messages(cont.)

```
ValidateTopic(topicValue)
{
    if(topicValue==='default')
  {
    this.topicHasErr=true;
  }
    else
  {
    this.topicHasErr=false;
  }
}
```

Then change **topic.valid** to !topicHasErr and change topic.invalid to topicHasErr

6-Displaying Error Messages(cont.)

```
<div class="form-group">
<select class="custom-select" #topic (blur)="ValidateTopic(topic.value)"
(change)="ValidateTopic(topic.value)"
required [class.is-invalid]="topicHasErr && topic.touched"
[(ngModel)]="userModel.topic" name="ddlTopic">
<option value="default">I'm Interested In</option>
<option *ngFor="let topic of
topics">{{topic}}</option>

</select>
<small class="text-danger" [class.d-none]="!topicHasErr ||
topic.untouched"> Please choose a topic</small>

</div>
```

6-Displaying Error Messages(cont.)

- **Form level validation**
- We need to disable form submit button until the form be valid.

```
<input type="submit" [disabled]="userForm.form.invalid"
     class="btn btn-primary" value="Submit Form">
```

- Note that userForm is a template reference variable of ngForm but note that is will fail in the select option so we need another condition.

- `<input type="submit" [disabled]="userForm.form.invalid || topicHasErr" class="btn btn-primary" value="Submit Form">`

TDF (Cont.) What we will do?

- Generate a new CLI project.
- Add the form HTML.
- Biding data.
- Tracking state and validity.
- Providing visual feedback.
- Displaying error messages.
- Posting data to the server.

Posting data to the server.

- Now we are going to learn how to post form data to the server.
- The first step is to add **novalidate** attribute to the form tag.
- <form #userForm="ngForm" novalidate>

- This will prevent from browser validation when we click on the submit button.

Posting data to the server(cont.)

- The next step is to bind to ngSubmit event when submit button clicked.
- <form #userForm="ngForm" novalidate **(ngSubmit)="onSubmit()">**
- The next is to define the **onSubmit** event handler.

Posting data to the server(cont.)

- Now to be able to send the data to the server we need to use a service.
- To generate a service
- ng g s enrollment

- Then the first step is to import HttpClient then inject it.
- `import { HttpClient } from '@angular/common/http';`
- `constructor( private _http:HttpClient)`

Posting data to the server(cont.)

- Then we need to import HttpClientModule in app.module.ts and add it to the imports array.
- `import {HttpClientModule} from '@angular/common/http';`

Posting data to the server(cont.)

- Then in enrollment.service.ts we now ready to post data
- Define property for example called _url which is the url we will post data to it.

- Make a method for example called enroll which will make the post request.

```
enroll(user:User)
{
return this._http.post<any>(this._url,user);
}
```

Posting data to the server(cont.)

- The post request will return a response as observable so we need to subscripe to the observable in app.commponent.ts
- First import the enrollment service and then inject it.
- `import { EnrollmentService } from '../enrollment.service';`
- **Inject it**
- `constructor( private _EnrollService:EnrollmentService)`
- `{}`

Posting data to the server(cont.)

Then we need to subscribe to the service

```
onSubmit()
{
//console.log(this.userModel);
this._EnrollService.enroll(this.userModel).
subscribe(
response => console.log('Success!',
response),
error => console.log('error',error)
)
}
```

Express Node Server

- First create new folder call it **Server**
- Initialize new package json file by the following command
- **npm init –-yes**
- Then lets install the dependencies by the following command
- **npm install --save express body-parser cors**
- **express** is a web server
- **Body-parser** is the middelware to handle form data
- **cors** is a package to make request across different ports

Express Node Server(cont.)

- Now after the dependencies installed in the JSON folder create a new file name it **server.js**
- Within this file begin with require these packages that we just install

```
const express = require('express');
const bodyParser=require('body-parser');
const cors=require('cors');
```

Express Node Server(cont.)

- Then create const for the port number that the express server will run on

```
const PORT=3000;
```

- Now create instance of express

```
const app=express();
```

Express Node Server(cont.)

- We also specify body parser to handle json data

```
app.use(bodyParser.json());
```

- We also need to use cors package

```
app.use(cors());
```

- Then lets add code to test get request

```
app.get('/',function(req,res){
res.send("hello from node server");
});
```

Express Node Server(cont.)

- Finally we listen to request on the specified port

```
app.listen(PORT,function(){
console.log("Server running on port
"+PORT)
});
```

- Start the node server and test

```
node server
```

Express Node Server(cont.)

- Now will add end point which our angular application will post data

```
app.post('/enrollment',function(req,res){
console.log(req.body);
res.status(200).send({"data":"Recieved successfully"});
});
```

Questions

Any Questions?

THANK YOU ☺