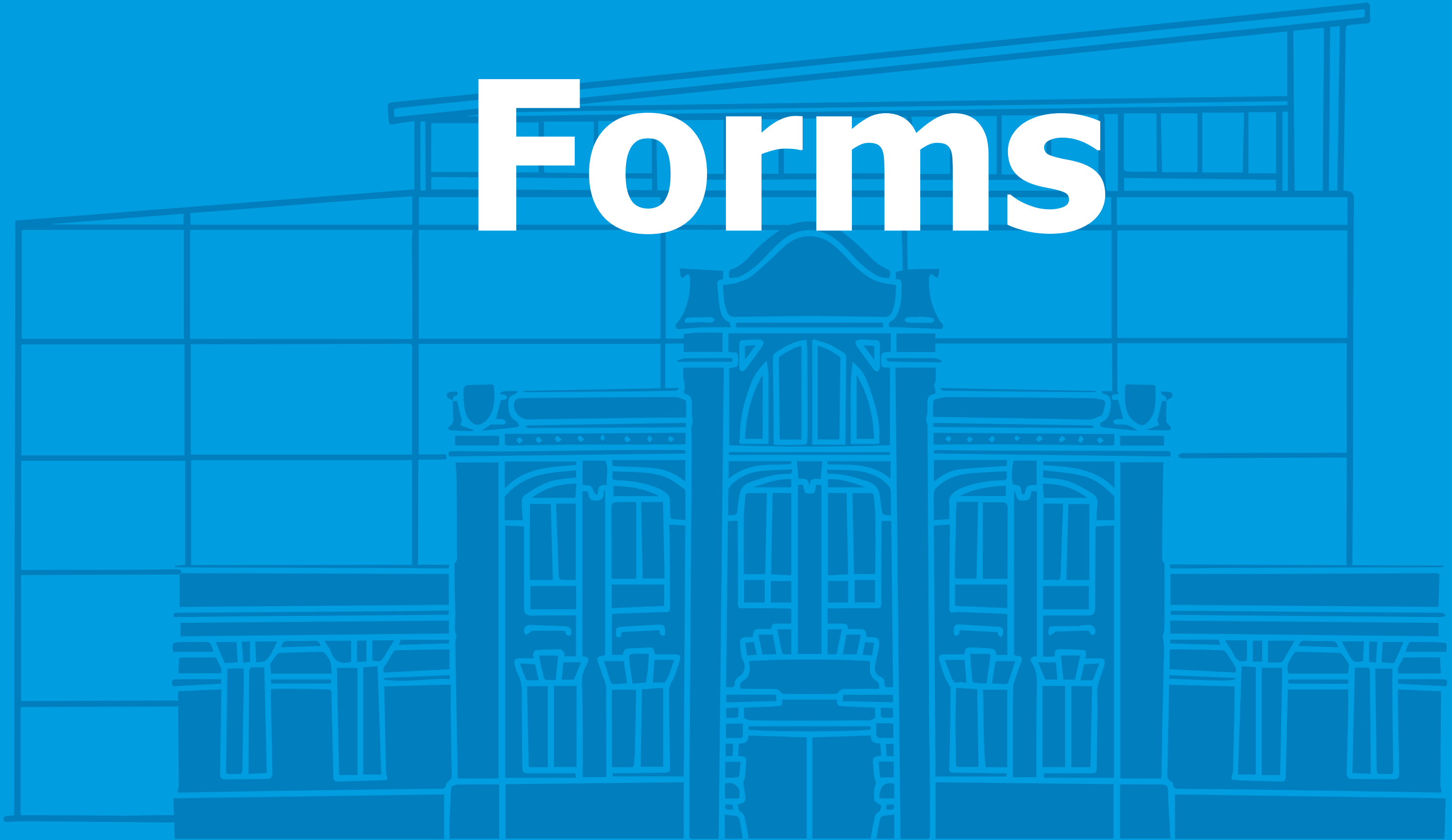
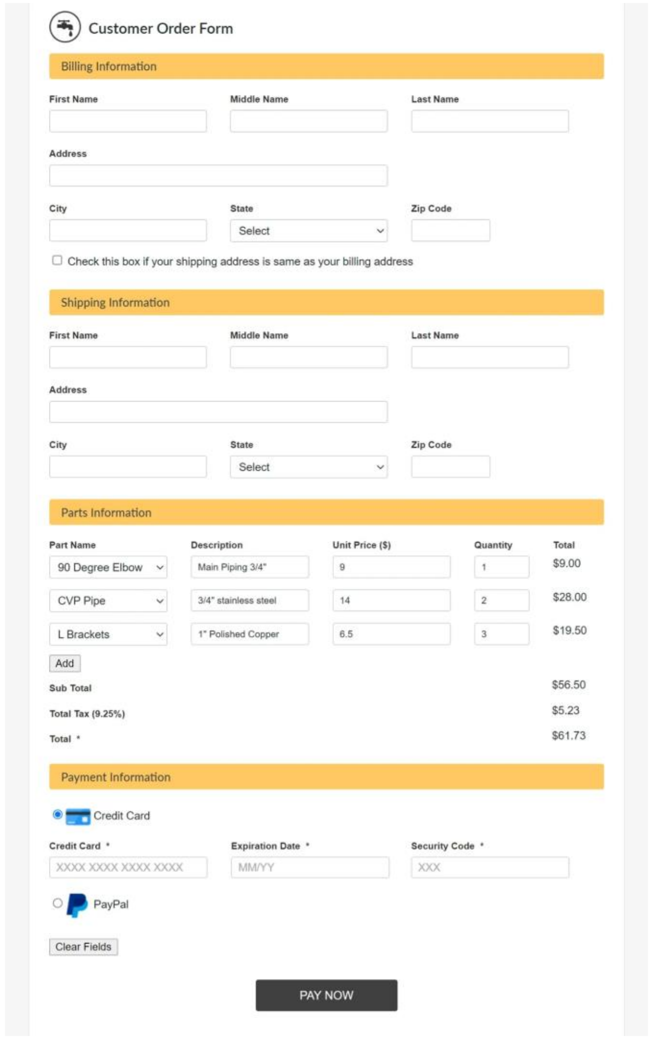
Angular 14 - 12

Forms



Forms in Angular



Customer Order Form

Billing Information

First Name	Middle Name	Last Name

Address

City	State	Zip Code
	Select	

☐ Check this box if your shipping address is same as your billing address

Shipping Information

First Name	Middle Name	Last Name

Address

City	State	Zip Code
	Select	

Parts Information

Part Name	Description	Unit Price ($)	Quantity	Total
90 Degree Elbow	Main Piping 3/4"	9	1	$9.00
CVP Pipe	3/4" stainless steel	14	2	$28.00
L Brackets	1" Polished Copper	6.5	3	$19.50

Add

Sub Total	$56.50
Total Tax (9.25%)	$5.23
Total *	$61.73

Payment Information

● Credit Card

Credit Card *	Expiration Date *	Security Code *
XXXX XXXX XXXX XXXX	MM/YY	XXX

○ PayPal

Clear Fields

PAY NOW

- **Forms can be very complex**
  - Inputs are intended to modify data, both on the page and on the server
  - Changes often need to be reflected elsewhere on the page
  - Users have a lot of leeway in what they enter, so values should be validated
  - The UI needs to clearly state expectations and errors, if any.
  - Dependent fields can have complex logic
  - We want to be able to test our forms, without relying on DOM selectors
- Fortunately, **Angular has tools to help** to deal with this complexity:
  - **FormGroups** and **FormControls** encapsulates the inputs (and groups) on our forms and gives us objects to work with.
  - **Validators** give us the ability to validate inputs the way we want.
  - **Observers** let us review changes to our formulary and act accordingly
  - Two approaches: Reactive Forms and Template-driven Forms.
  - https://angular.io/guide/forms-overview

FormControls and FormGroups

FormControl

- A FormControl represents a single input field: it is the smallest unit of a form.
- FormControls encapsulates the value of the field, and states such as being valid, modified, or having errors.

```
nameControl = new FormControl();

// we can access the control to check its state
name = nameControl.value;
nameControl.errors // -> StringMap<string, any> of errors
nameControl.dirty  // -> false
nameControl.valid  // -> true
```

```
// template
...
<input type="text" [formControl]="name" />
...
```

- To **create** forms, we create FormControls (and groups of FormControls) and then assign metadata and logic to them.

FormControls and FormGroups

FormGroup

- Most forms have more than one field, so we need a way to manage multiple FormControls. For example, if we wanted to check the validity of our form, it's cumbersome to iterate over an array of FormControls and check the validity of each FormControl.
- FormGroups solve this problem by providing a wrapper interface around a collection of FormControls.

```
userInfo = new FormGroup ({
    firstName : new FormControl ( "Nate" ),
    lastName : new FormControl ( "Murray" ),
    cp : new FormControl ( "90210" )
})
```

FormControls and FormGroups

- FormGroup and FormControl have a common ancestor (AbstractControl). That means we can check the state or value of group just as easily as a single FormControl:

```
personInfo.value ; // -> { // firstName: " John ", // lastName: " Perez ", // cp : "90210" // }

// We can access the state of the group
// Values depend on the state of the array of FormControl s that compose it :
personInfo.errors // -> StringMap<string, any> of errors
personInfo.dirty  // -> false
personInfo.validate  // -> true // etc.
```

Model Driven/Reactive Forms

- First we'll need to import the **ReactiveFormsModule** module , into our NgModule.
- ReactiveFormsModule contains the classes and directives that we will need to work with reactive forms.

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

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

Model Driven/Reactive Forms

- Whether we're in the model-driven or template-driven form, we need an HTML form.
- We will set up the typical registration form:

```
<form novalidate>
    <fieldset>
        <div class="form-group">
            <label>Name</label>
            <input type="text" class="form-control">
        </div>

        <div class="form-group">
            <label>Last Name</label>
            <input type="text" class="form-control">
        </div>
    </fieldset>

    <div class="form-group">
        <label>Email</label>
        <input type="email" class="form-control">
    </div>

    <div class="form-group">
        <label>Password</label>
        <input type="password" class="form-control">
    </div>

    <div class="form-group">
        <label>Language</label>
        <select class="form-control">
            <option value="">Please select a language</option>
        </select>
    </div>
</form>
```

Model Driven/Reactive Forms

- We represent the form as a model made up of instances of FormGroups and FormControls .
- We create the model for our form in our component:

```
import { FormControl, FormGroup } from '@angular/forms';

@Component()
export class NewProductComponent implements OnInit {
  myform: FormGroup | null = null;

  ngOnInit(): void {
    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: new FormControl(),
        lastName: new FormControl(),
      }),
      email: new FormControl(),
      password: new FormControl(),
      language: new FormControl(),
    });
  }


}
```

Model Driven/Reactive Forms

- **myform** is an instance of a FormGroup class and represents our form itself.
- Each **form control** in the template is represented by an **instance of FormControl**. This encapsulates the state of the control, such as whether it is valid or invalid and even its current value.
- These **FormControls instances nest** within our top level myform: FormGroup , but the interesting thing is that we can nest **FormGroups within other FormGroups**.
  - In our model, we have grouped the firstName and lastName controls under a name called FormGroup that is nested in our top level myform: FormGroup.
  - Like the FormControl instance, FormGroup FormGroup instances encapsulate the state of all its inner controls, for example, an instance of a FormGroup is only valid if all its inner controls are also valid.

Model Driven/Reactive Forms

- We have the HTML template for our form and the form model in our component, now we need to link them together.
- We associate the form through the **formGroup** directive

```
<form [formGroup]="myform"> ... </form>
```

- formControlName directive to bind each form control in the template to a named form control in the model:

```
<div class = "form-group">
    <label> Email </ label>
    <input type = "email" class = "form-control" formControlName = "email"   required>
</div>
```

- In the template, the controls that we want to group must be surrounded by another element. This happens with the name.

```
<fieldset formGroupName = "name"> ... </fieldset>
```

Model Driven/Reactive Forms: Validation - Validators

- Validators are rules that a form input must follow. If the input does not match the rule, then the control is invalid.
- For example in our signup form example, most of the fields should be required and we would also like to specify some more complex validators on the password field to make sure the user enters a good strong password.
- We can apply validators by adding attributes to the template or by defining them on our FormControls in our model.
- Angular comes with a set of pre-built validators to match which we can define via the standard HTML 5 attributes, i.e. required , minlength , maxlength and pattern which we can access from the Validators module.
- https://angular.io/guide/form-validation

Model Driven/Reactive Forms: Validation - Validators

- In our example we will import Validators and associate the validators to each formControl.

```
import { FormControl, FormGroup, Validators } from
'@angular/forms';

@Component()
export class NewProductComponent implements OnInit {
  myform: FormGroup = new FormGroup({});

  ngOnInit(): void {
    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: new FormControl('', Validators.required),
        lastName: new FormControl('', Validators.required),
      }),
      email: new FormControl('', [
        Validators.required,
        Validators.pattern('[^ @]*@[^ @]*'),
      ]),
      password: new FormControl('', [
        Validators.minLength(8),
        Validators.required,
      ]),
      language: new FormControl(),
    });
  }
}
```

Model Driven/Reactive Forms: Validation - Validators

- There may be form fields that require more complex or custom rules for validation. In those situations, you can use a custom validator.

- Custom validators are defined with functions. It can exist as a function in a component file directly. Otherwise, if the validator needs to be reused, it can exist in a separate service.

```
  ValidateLanguage = (control: AbstractControl) => {
    if (!(control.value.length >= 2) || control.value.includes('*')) {
      return { invalidLanguage: true };
    }
    return null;
  };
```

```
...
language: new FormControl('',[this.ValidateLanguage])
...
```

Model Driven/Reactive Forms: Validation - Control Status

- The form control instance in our model **encapsulates status about the control**, such as whether it is valid or has been touched.
- We can get a reference to these form control instances in our template via the controls property of our myform model.
  - **dirty** will tell us if the user has changed the value of the control
  - **pristine** indicates whether the value of the control is pristine

```
<pre>Dirty? {{ myform.controls['email'].dirty }}</pre>
<pre>Pristine? {{ myform.controls['email'].pristine }}</pre>
```

  - **touched** indicates whether the user touched the control (even if the value has not changed)

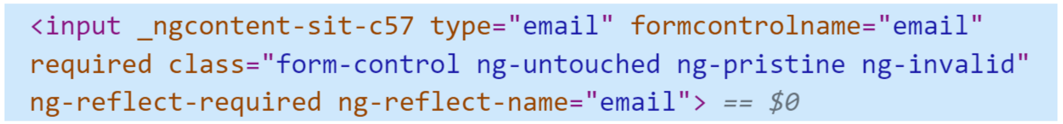
```
<pre>Touched? {{ myform.controls['email'].touched }}</pre>
```

  - **valid** indicates if the field is valid (it has passed all validators)

```
<pre>Valid? {{ myform.controls['email'].valid }}</pre>
```

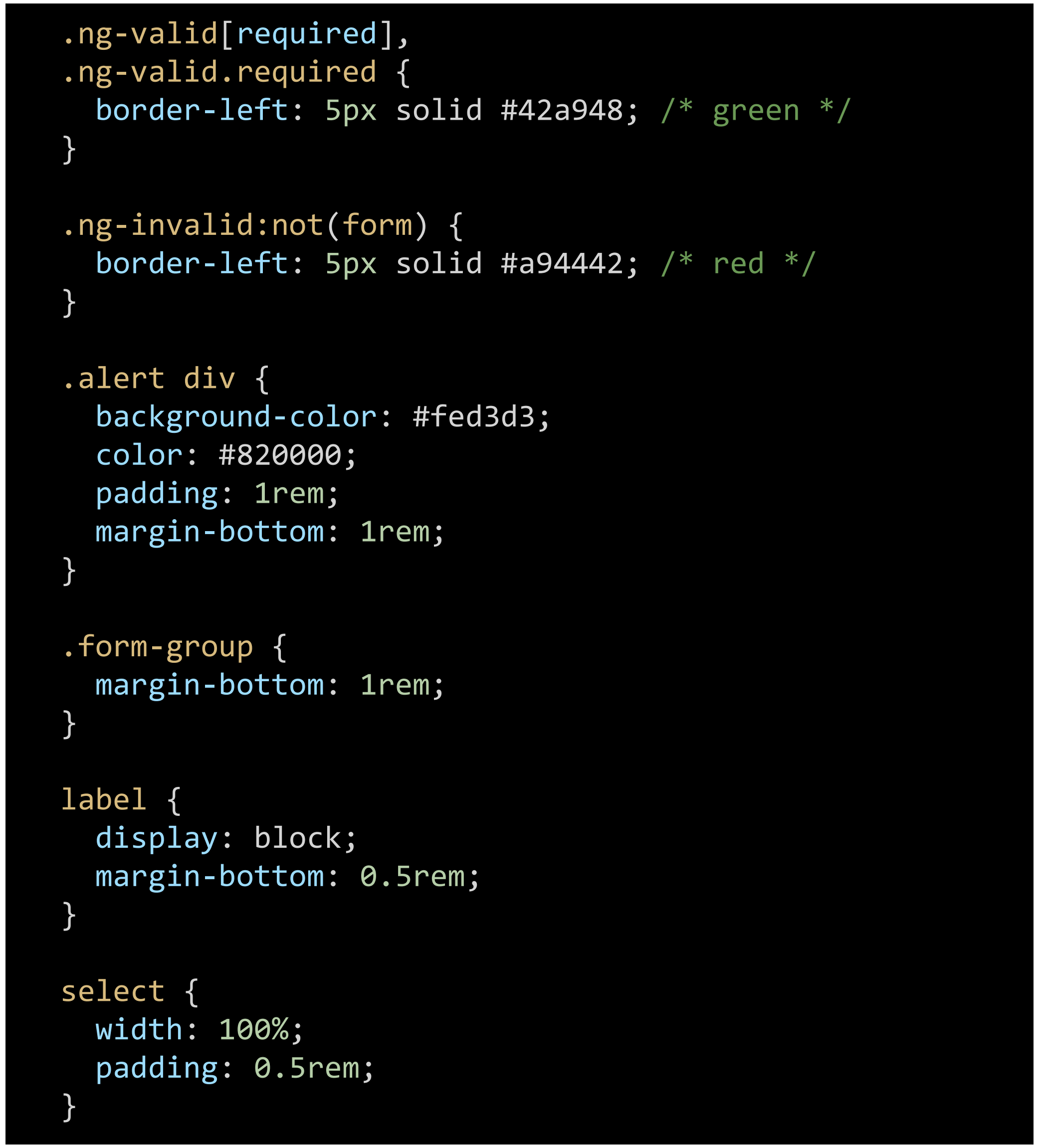
Model Driven/Reactive Forms: Validation - styling

- Angular automatically mirrors many control properties onto the form control element as CSS classes. Use these classes to style form control elements according to the state of the form. The following classes are currently supported.
    - .ng-valid
    - .ng-invalid
    - .ng-pending
    - .ng-pristine
    - .ng-dirty
    - .ng-untouched
    - .ng-touched
    - .ng-submitted (enclosing form element only)



```
<input _ngcontent-sit-c57 type="email" formcontrolname="email"
required class="form-control ng-untouched ng-pristine ng-invalid"
ng-reflect-required ng-reflect-name="email"> == $0
```

- We can define css styles for this classes.



```
.ng-valid[required],
.ng-valid.required {
  border-left: 5px solid #42a948; /* green */
}

.ng-invalid:not(form) {
  border-left: 5px solid #a94442; /* red */
}

.alert div {
  background-color: #fed3d3;
  color: #820000;
  padding: 1rem;
  margin-bottom: 1rem;
}

.form-group {
  margin-bottom: 1rem;
}

label {
  display: block;
  margin-bottom: 0.5rem;
}

select {
  width: 100%;
  padding: 0.5rem;
}
```

Model Driven/Reactive Forms: Validation - styling

- We can **combine css classes** (for example from boostrap) with FormControl dirty and invalid properties and the ngClass directive to give the user visual feedback

```
<input type="email" class="form-control" formControlName = "email"   required [ngClass]="{
                'is-invalid': myform.controls['email'].invalid && myform.controls['email'].dirty,
                'is-valid': myform.controls['email'].valid && myform.controls['email'].dirty
              }">
```

- The above can quickly become cumbersome, especially for nested controls. For this we can use local properties to help us

```
  myform: FormGroup = new FormGroup({});
  firstName = new FormControl('', Validators.required);
  lastName = new FormControl('', Validators.required);

    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: this.firstName,
        lastName: this.lastName
      }), ...
    });
```

```
<input type="text" class="form-control" formControlName = "firstName"   required  [ngClass]="{
                 'is-invalid': firstName.invalid && firstName.dirty,
                 'is-valid': firstName.valid && firstName.dirty
               }">
```

Model Driven/Reactive Forms: Validation - messages

- In the same way we can display validation messages by combining styles, control state and Angular directives

```
<div class="form-group">
    <label>Password</label>
    <input type="password" class="form-control" formControlName="password">
    <div class="form-control-feedback" *ngIf="password.invalid && password.dirty">
          Invalid password
    </div>
</div>
```

- Likewise, we can go into more detail by reviewing the **errors** property to contrast it with the validators.

```
<div class="form-control-feedback" *ngIf="password.errors && (password.dirty || password.touched)">
    <p *ngIf="password.errors.required">Password is mandatory</p>
    <p *ngIf="password.errors.minlength">Password must have 8 characters</p>
</div>
```

Model Driven/Reactive Forms: Submit and reset

- To capture the submit event we will use Angular's events directive with **ngSubmit** and call a process function.

```
<form (ngSubmit)="onSubmit()">
    ...
    ...
    ...
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

- In the component, the function will be defined.
- Also, if we want to clear the form after processing, we can use the **reset** method .

```
  onSubmit() {
    if (this.myform.valid) {
      console.log("Form Submitted!");
      this.myform.reset();
    }
  }
```

Reactive model

- Both FormControls and FormGroups expose an observable value called **valuesChanged**. By subscribing to this **observable**, we can react in real time to changing values of an individual form control, or a group of them.
- A use case could be the implementation of a lookup field in an application that communicates with an API.
  - Since calling an API is relatively expensive, we want to limit the number of API calls to only when absolutely necessary.

```
<input type="search" class="form-control" placeholder="Please enter search term" [formControl]="searchField">
<hr />
<ul>
    <li *ngFor="let search of searches">{{ search }}</li>
</ul>
```

```
   searchField = new FormControl();
   searches: string[] = [];
```

Reactive model

- To react to changes in this form, we need to subscribe to the valueChanges observable of our searchField field:

```
searchField = new FormControl();
searches: string[] = [];

ngOnInit(): void {
  this.searchField.valueChanges.subscribe((term) => {
    this.searches.push(term);
  });

}
```

- Ideally, we only make a request when the user has stopped typing and the term is different. RxJS has an operator that implements it called **debounceTime** and **distinctUntilChanged**:

```
searchField = new FormControl();
searches: string[] = [];

ngOnInit(): void {
  this.searchField.valueChanges.pipe(
    debounceTime(400),
    distinctUntilChanged()
  ).subscribe((term) => {
      console.log('term:',term);
      this.searches.push(term);
    }).;
}
```

Template-driven Forms

- The key to this approach is to understand that it uses the same models as the model-driven approach. The difference is that in this one, Angular creates the models, FormGroups and FormControls from the directives that we add in the template.
- The directives we'll need are in the FormsModule in the **FormsModule** module (not ReactiveFormsModule), so let's import it and add it to our NgModule.

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms’;
@NgModule({
  declarations: [...],
  imports: [..., FormsModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

- In the template we will use the **ngForm** , **ngModel** and **ngModelGroup** directives to define a form, a formControl and a formGroup respectively

```
<form #f="ngForm"> ...
    <input name="foo" ngModel>
</form>
```

Template-driven Forms

- Another feature of the ngModel directive is that it allows us to configure bidirectional binding.
- This, together with a domain model expressed in a class, greatly facilitates data processing.

```
export class Signup {
  constructor(
    public name: string = '',
    public surname: string = '',
    public email: string = '',
    public password: string = '',
    public language: string = ''
  ) {}
}
```

- We will use the domain model to instantiate our component

```
model:Signup = new Signup();
```

- What we will use to associate to the form

```
<input ... [(ngModel)]="model.email" >
```

Template-driven Forms: Validation

- In this approach we define the validators via HTML5 directives and attributes in our template.
- For example, if all fields are required, we will simply add the **required** attribute.
- We can also use a validation pattern via **pattern**.
- Also minlength and maxlength… definitely the **same attributes as HTML5**.

```
...
<input type="email"
     class="form-control"
     name="email"
     [(ngModel)]="model.email"
     required
     pattern="[^ @]*@[^ @]*" />
...

<input type="password"
       class="form-control"
       name="password"
       [(ngModel)]="model.password"
       required
       minlength="8" />
...
```

Template-driven Forms: Validation

- We can also access the state of each control by going through the top level form group since the **ngForm** directive makes the top level FormGroup available via the **.form** property.
- Therefore we can show the valid, dirty or touched status of our fields

```
<pre>Valid? {{f.form.get('email')?.valid}}</pre>
<pre>Dirty? {{f.form.get('email')?.dirty}}</pre>
<pre>Touched? {{f.form.get('email')?.touched}}</pre>
```

- For example in the email field

```
<input class="form-control" [(ngModel)]="model.email" name="email" required pattern="[^ @]*@[^ @]*" placeholder="email"
[ngClass]="{'is-invalid': f.form.get('email')?.invalid && (f.form.get('email')?.dirty || f.form.get('email')?.touched),
                'is-valid': f.form.get('email')?.valid && (f.form.get('email')?.dirty || f.form.get('email')?.touched)
           }">
```

Template-driven Forms: Validation

- The NgForm directive provides us with a shortcut to the controls property so we can write **f.controls.email?.valid** instead of **f.form.controls.email?.valid**
- But both are still quite long and if we wanted to access a nested form control, it can be even more cumbersome:
  - f.controls.name.firstName?.valid
- ngModel provides us with a much shorter alternative. We can access the instance of our ngModel directive using a local template reference variable using the **#alias_field = ngModel** directive

```
<input ... [(ngModel)]="model.email" #email="ngModel"> </input>
```

- In our example

```
<input class="form-control" [(ngModel)]="model.email" name="email" required pattern="[^ @]*@[^ @]*"
                placeholder="email" #email="ngModel" [ngClass]="{'is-invalid': email.invalid && (email.dirty || email.touched),
                'is-valid': email.valid && (email.dirty || email.touched)
            }">
```

Template-driven Forms: Validation

- Likewise for form validation messages, we'll use the exact same method that we use on model-driven forms.
- As long as we name the local reference variables the same as the form controls in the model-driven approach, we can use the exact same HTML in our template-based forms:

```
<div class="form-control-feedback" *ngIf="email.errors && (email.dirty || email.touched)">
    <p *ngIf="email.errors.required">Email is required</p>
    <p *ngIf="email.errors.minlength">Email must contain at least the @ character</p>
</div>
```

Template-driven Forms: Custom Validator

- Building a Validator in template-driven forms is similar to building an **Angular directive**.
- The directive must implement the **Validator interface**.
- We need to register it in Angular **Providers** using **NG_VALIDATORS**. This adds our validator to the collection of all existing validators
- We also set **multi:true** because there can be more validation directives. **useClass** provides the class name of our custom directive.

Template-driven Forms: Custom Validator

```
import { Directive } from '@angular/core';
import { Validator,  NG_VALIDATORS,  ValidatorFn,  FormControl,} from '@angular/forms';

@Directive({
  selector: '[appEmailvalidator]',
  providers: [{provide: NG_VALIDATORS, useClass: EmailvalidatorDirective, multi: true,}],
})
export class EmailvalidatorDirective implements Validator {
  validator: ValidatorFn;
  constructor() {
    this.validator = this.emailValidator();
  }

  validate(c: FormControl) {
    return this.validator(c);
  }

  emailValidator(): any {
    return (control: FormControl) => {
      if (control.value != null && control.value !== '') {
        let isValid =/^[_a-z0-9]+(\.[_a-z0-9]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$/.test(control.value);
        if (isValid) {
          return null;
        } else {
          return {emailvalidator: { valid: false }};
        }
      } else return null;
    };
  }
}
```

```
<input class="form-control" [(ngModel)]="model.email" name="email" required appEmailvalidator
                 placeholder="email" #email="ngModel" [ngClass]="{'is-invalid': email.invalid && (email.dirty || email.touched),
                 'is-valid': email.valid && (email.dirty || email.touched)
             }">
```

Template-driven Forms: Submit and reset

- We'll do it the same as in the model-driven case, but in order to access the form's model we'll need to use the **ViewChild** decorator.
- We will use the model instance to process it in the submit callback function

```
<form #f="ngForm" (ngSubmit)="onSubmit()">... </form>
```

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

@Component()
export class NewProductComponent implements OnInit {
  model: Signup = new Signup();
  @ViewChild('f') form: any;

  onSubmit() {
    if (this.form.valid) {
      console.log('Form Submitted!');
      this.form.reset();
    }
  }
}
```

Let's put it into practice: Tasks/Projects App

1. Create a route that allows you add a new task.
2. Same for projects
3. Create another component for adding or removing users to tasks. It must also be reusable for projects and be reactive.

Next steps

We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

Thanks!

Follow us: