Agenda

- Model Driven / Reactive Forms
- Form with **@ViewChild**
- Validations
- Custom Validators
- Submitting and Resetting forms

Model Driven / Reactive Forms

Model driven forms are more **powerful and easy** to do functionalities which are complex when using template driven forms.

Step1: Create a simple form

File: **mytemplate.html**

```
<form novalidate>
        <div class="form-group">
          <label>First Name</label>
          <input type="text" class="form-control required" />
        </div>
        <div class="form-group">
          <label>Last Name</label>
          <input type="text" class="form-control" required />
        </div>
        <pre>{{myform.value | json}}</pre>
</form>
```

Step2: Create a form component and include form models

To create the instances of our form with form controls inside we need to use **FormGroup, FormControl.**

- **FormGroup**: Creates the form instance
- **FormControl:** Creates the form control in your template

File: **form.component.ts**

```
import { Component, OnInit, Pipe } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';


@Component({
   selector: 'my-form',
   templateUrl: "./mytemplate.html",
})
export class FormComponent implements OnInit {
```

```
  myform: FormGroup; //FormGroup  is a dictionary of FormControls
  firstName: FormControl;
  lastName: FormControl;
  message = "";

  ngOnInit() {
    this.createFormControls();
    this.createForm();
  }
  createFormControls() {
    this.firstName = new FormControl();
    this.lastName = new FormControl();
  }
  createForm() {
    this.myform = new FormGroup({
      firstName: this.firstName,
      lastName: this.lastName,
    });
  }
}
```

Step3: Linking form controls to form template

Here we link our **myform: FormGroup** to the template **<form>** tag and **firstName:FormControl and lastName:FormControl** to the template form controls.

- **[formGroup]:** This lets our component know that this is the form associated with **myform**
- **[formControlName]:** This directive is used to map each form control of our form with the form controls in the component.

File: **mytemplate.html**

```
<form [formGroup]="myform" novalidate>
  <div class="form-group">
    <label>First Name</label>
    <input type="text" class="form-control" formControlName="firstName" required />
  </div>
  <div class="form-group">
    <label>Last Name</label>
    <input type="text" class="form-control" formControlName="lastName" required />
  </div>
  <pre>{{myform.value | json}}</pre>
```

```
</form>
```

File: **app.component.ts**

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

@Component({
  selector: 'my-app',
  template: '<my-form></my-form>'
})
export class AppComponent { }
```

File: **app.module.ts**

Here we need to import **ReactiveFormsModule** in the root **@NgModule** to make use of directives that comes from this library.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule, FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { FormComponent } from './app.FormComponent';

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

Step 4: Run and test the application.

Validations

Step4: Providing validations

Angular provides us built-in validators via standard HTML5 attributes such as

- **Required**
- **Maxlength**
- **Minlength**
- **Pattern**

Angular provides built-in attributes to know the state of the form and its form-controls

- Track change-state and validity with **ngModel**

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

And our **FormContol( )** can be overloaded by providing the Validators

- **first parameter**: **Initial value** of the control
- **second parameter**: It contains either a **single validator or list of validators**

Modify the above **FormComponent** as follows,

- firstName Validations**:** (**required**, **pattern** which allows only alpha numeric but not special characters)
- lastName Validations: (**required**, **maxlength** of 10 characters)

```
createFormControls() {
    this.firstName = new FormControl('', [Validators.required, Validators.pattern('[a-zA-Z0-9 ]+')]);
    this.lastName = new FormControl('', [Validators.required, Validators.maxLength(10)]);
  }
```

Now we need to display the **validation message** to the form controls

Add this div under the input tag

```
<div class="form-control-feedback" *ngIf="firstName.errors && (firstName.dirty || firstName.touched)">
            <p *ngIf="firstName.errors.required">First Name is required</p>
            <p *ngIf="firstName.errors.pattern">First Name cannot contain special charaters (@, _, etc.,) </p>
</div>
<div class="form-control-feedback" *ngIf="lastName.errors && (lastName.dirty || lastName.touched)">
            <p *ngIf="lastName.errors.required">Last Name is required</p>
            <p *ngIf="lastName.errors.maxlength">Last Name length can be only 10 characters long</p>
</div>
```

~~**Validation Styling:**~~

~~Bootstrap has classes for showing visual feedback for form controls when they are invalid. For instance, if we add the has-danger class to the parent div of the input control with the class of form-group it adds a red border. Conversely if we add the has-success class it adds a green border.~~

~~Add the following code to the ** ** tag which contains the **form controls**~~

```
<div class="form-group" [ngClass]="{
    'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),
    'has-success': firstName.valid && (firstName.dirty || firstName.touched)
```

```
}">
```

Output:

First Name

Last Name

```
{
  "firstName": null,
  "lastName": null
}
```

Submitting and Resetting form

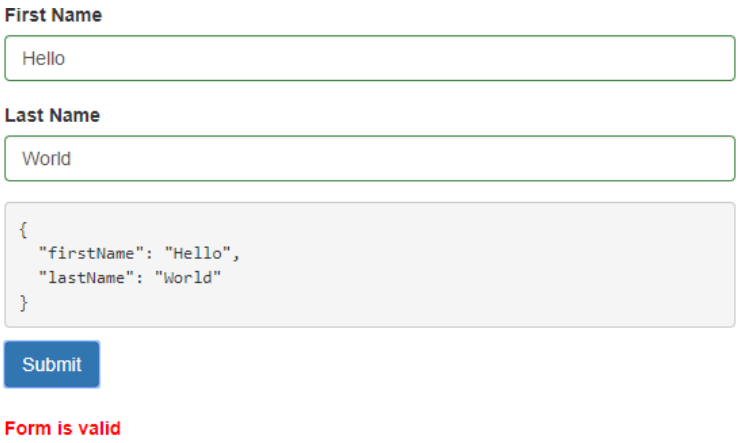
Create a method **onSubmit( )** in the **FormComponent**

```
export class FormComponent implements OnInit {
   myform: FormGroup;
   firstName: FormControl;
   lastName: FormControl;
   message = "";
.
.
.
onSubmit() {
      if (this.myform.valid)
        this.message = "Form is valid";
      else
        this.message = "Form is invalid";
   }
}
```

From the above example add a button to submit the form, now if the form is valid it will show message **"Form is valid"** else it shows **"Form is invalid"**

```
<div *ngIf="message!='' style="color:red"><b>{{message}}</b></div>
<form [formGroup]="myform" (ngSubmit)="onSubmit()" novalidate>
          //. . .
          <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

```
<br />
```



Now let's send the form data to the **onSubmit(myform.value)**

```
<form [formGroup]="myform" (ngSubmit)="onSubmit(myform.value)" novalidate>
        //. . .
<form>
```

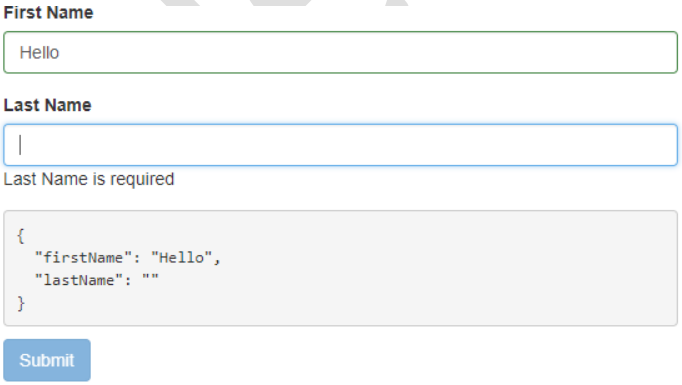
Do the following modification in **FormComponent.ts**

File: **app.FormComponent.ts**

```
onSubmit(form: any) {
    this.message = "Hello " + form.firstName + " " + form.lastName;
}
```

Now, **disable the submit button if form is invalid**

```
<button type="submit" class="btn btn-primary" [disabled]="!myform.valid">Submit</button>
```



Resetting form data: To reset the form in a model driven form we just need to call the **reset().**

```
onSubmit(form: any) {
    this.message = form.firstName + " " + form.lastName;
```

```
    this.myform.reset();
}
```

Reactive Form

In angular **FormControls and FormGroups** exposes an **observable called valuesChanged**, by **subscribing** to this observable we can react to the changes of the form control or group of form controls.

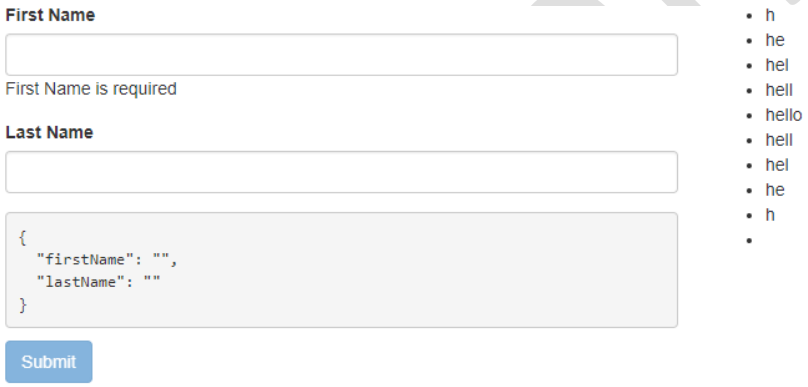
Rxjs library provides operators such as **debounceTime, distinctUntilChange** etc.

From the above example, add the following in **ngOnInit()**

```
ngOnInit() {
    this.createFormControls();
    this.createForm();

    this.firstName.valueChanges.subscribe(change => {
                                        this.changes.push(change);
                                    });
  }
```

This will detect on every key press event.



Now if we want to detect the changes when the user stopped typing they we can make use of **debounceTime()** **RXJS operator.** For that we need to import the required operator.

```
import 'rxjs/add/operator/debounceTime';
ngOnInit() {
    this.createFormControls();
    this.createForm();

    this.firstName.valueChanges
      .debounceTime(400)
      .subscribe(change => {
      this.changes.push(change);
```

```
    });
  }
```

Now if we want to detect only when user make any changes we can make use of **distinctUntilChanged** operator

```
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
ngOnInit() {
      this.createFormControls();
      this.createForm();
      this.firstName.valueChanges
         .debounceTime(400)
         .distinctUntilChanged()
         .subscribe(change => {
           this.changes.push(change);
         });
}
```

Using FormBuilder

We can use the **FormBuilder** to create the form. We need to import **FormBuilder** from forms library.

File: **app.FormComponent.ts**

```
import { Component, OnInit, Pipe } from '@angular/core';
import { FormGroup, FormControl, Validators, FormBuilder } from '@angular/forms';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';

@Component({
   selector: 'my-form',
   templateUrl: `./template.html`,
})
export class FormComponent implements OnInit {
   constructor(private fb: FormBuilder) { }
   myform: FormGroup;
   message = "";
   changes:string[] = [];
```

```
  ngOnInit() {
    this.myform = this.fb.group({
      firstName: ['', [Validators.required, Validators.pattern('[a-zA-Z0-9 ]+')]],
      lastName: ['', [Validators.required, Validators.maxLength(10)]]
    });
    this.myform.get("firstName").valueChanges
      .debounceTime(400)
      .distinctUntilChanged()
      .subscribe(change => {
      this.changes.push(change);
    });
  }
  onSubmit(form: any) {
    this.message = form.firstName + " " + form.lastName;
    this.myform.reset();
  }
}
```

To provide validations we need to follow a different syntax as follows.

```
myform.controls['firstName'].invalid
myform.controls['firstName'].valid
myform.controls['firstName'].touched
```

File: **template.html**

```
<div class="container">
  <div class="col-xs-6">
    <form [formGroup]="myform" (ngSubmit)="onSubmit(myform.value)" novalidate>
      <div class="form-group"  [ngClass]="{
                  'has-danger': myform.controls['firstName'].invalid && (myform.controls['firstName'].dirty ||
myform.controls['firstName'].touched),
                  'has-success': myform.controls['firstName'].valid && (myform.controls['firstName'].dirty ||
myform.controls['firstName'].touched)
                  }">
        <label>First Name</label>
        <input type="text" class="form-control" formControlName="firstName" required />
        <div class="form-control-feedback" *ngIf=" myform.controls['firstName'].errors &&
(myform.controls['firstName'].dirty || myform.controls['firstName'].touched)">
```

```
            <p *ngIf=" myform.controls['firstName'].errors.required">First Name is required</p>
            <p *ngIf=" myform.controls['firstName'].errors.pattern">First Name is invalid</p>
          </div>
</div>
        <div class="form-group" [ngClass]="{
      'has-danger': myform.controls['lastName'].invalid && (myform.controls['lastName'].dirty ||
myform.controls['lastName'].touched),
      'has-success': myform.controls['lastName'].valid && (myform.controls['lastName'].dirty ||
myform.controls['lastName'].touched)
    }">
          <label>Last Name</label>
          <input type="text" class="form-control" formControlName="lastName" required />
          <div class="form-control-feedback" *ngIf=" myform.controls['lastName'].errors &&
(myform.controls['lastName'].dirty || myform.controls['lastName'].touched)">
            <p *ngIf=" myform.controls['lastName'].errors.required">Last Name is required</p>
            <p *ngIf=" myform.controls['lastName'].errors.maxlength">Last Name length can be only 10 characters
long</p>
          </div>        </div>
        <pre>{{myform.value | json}}</pre>
        <button type="submit" class="btn btn-primary" [disabled]="!myform.valid">Submit</button>
      </form>
      <br />
      <div *ngIf="message!='' " style="color:red"><b>{{message}}</b></div>
    </div>
    <div class="col-xs-6" *ngIf="changes?.length > 0">
      <ul>
        <li *ngFor="let change of changes">{{change}}</li>
      </ul>
    </div>
</div>
```