

# FORMS

---

# Forms are Crucial, Forms are Complex

- In Angular, there are 2 approaches to handle user's input through forms:
  - Reactive forms ((also known as model-driven)
  - Template-driven forms

# Template Driven Forms vs Angular Reactive Forms

- They use their very own modules: [ReactiveFormsModule](#) and [FormsModule](#).
- Reactive forms are synchronous while Template-driven forms are asynchronous.
- Template-driven forms are simple forms that can be made rather quickly. Reactive forms are more complex forms that give you greater control over the elements in the form.

Template-Driven Form	Reactive Form
Template-Driven Form is less explicit, and it is mainly created by Directives.	Reactive Form is more explicit and normally created within the Component class.
It supports the unstructured data model	It always supports the structured data model.
It uses directives for implementing Form validations	It uses the function for implementing Form Validations
More control over validation logic; good for complex forms	Good for simple forms
When form control value changes, it provides an asynchronous mechanism to update form controls.	When form control value changes, it provides synchronous mechanism to update form controls.

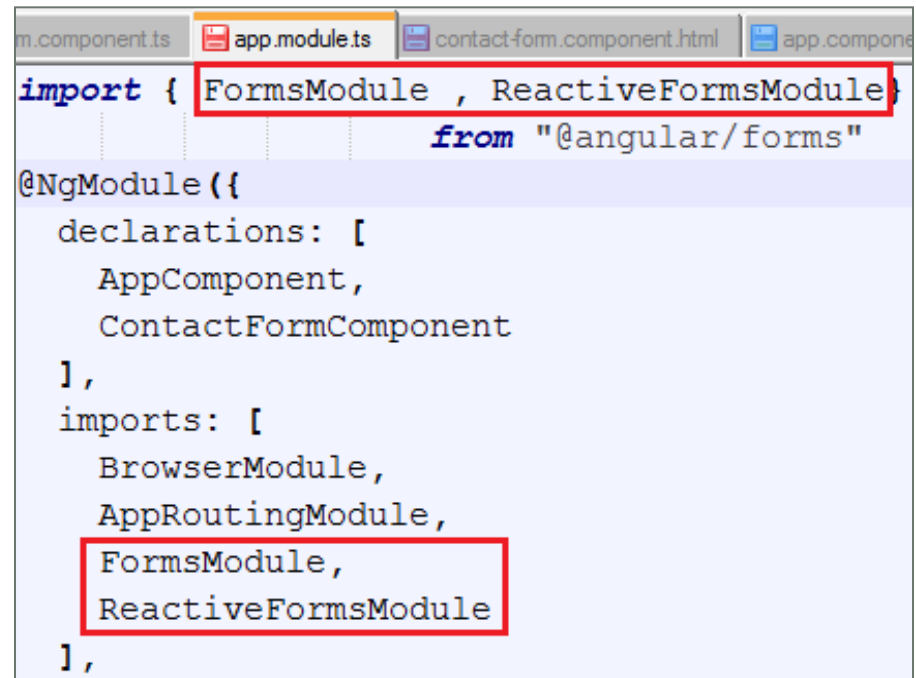
# Template Driven Forms

- Template-driven are those forms where we can write logic, validations, controls, etc. in the HTML template of the component.
  - The Template is totally responsible for setting the form elements in the UI, for implementing the validation using form control, etc.
  - In the end, the Template itself takes care of binding the values with the model and the form validation.
- Benefits of this approach:
  - It is much easier to use
  - This technique works perfectly in simple scenarios
  - It totally depends on two-way data binding techniques i.e. ngModel syntax.
  - It requires a minimum of code in the component part since most of the work is done in the HTML template part.
- Drawbacks:
  - This technique fails when we want to design some complex form in the UI section

# Forms

- Template driven forms
  - First, make sure that the **FormsModule** is imported in your app
  - FormsModule gives us template driven directives like: **ngModel** and **NgForm**
- Reactive forms:
  - Import **ReactiveFormsModule**.
  - ReactiveFormsModule gives us directives like **formControl** and **ngFormGroup**

With a template driven form, most of the work is done in the template; and with the model driven form, most of the work is done in the component class.



```
import { FormsModule, ReactiveFormsModule }
      from "@angular/forms"
@NgModule({
  declarations: [
    AppComponent,
    ContactFormComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule
  ],
```

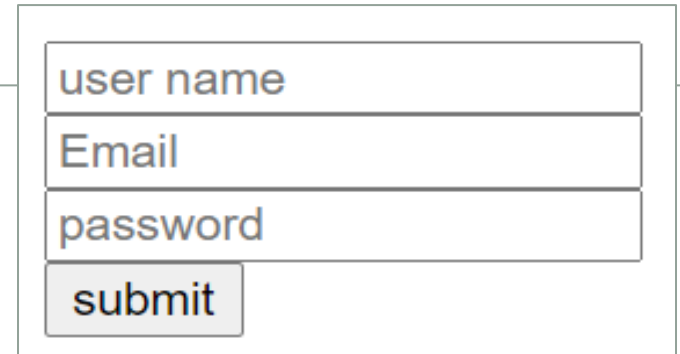
# Build a template-driven form

- Template-driven forms rely on directives defined in the FormsModule.
  - The **NgModel** directive reconciles value changes in the attached form element with changes in the data model, allowing you to respond to user input with input validation and error handling.
  - The **NgForm** directive creates a top-level **FormGroup** instance and binds it to a `<form>` element to track aggregated form value and validation status.
    - As soon as you import FormsModule, this directive becomes active by default on all `<form>` tags. You don't need to add a special selector.
  - The **NgModelGroup** directive creates and binds a FormGroup instance to a DOM element.

# Example

- Form structure for our form:

```
<form >  
  <input name = "uname" placeholder = "user name"> <br/>  
  <input type = "email" name = "email" placeholder = "Email"> <br/>  
  <input type = "password" name = "passwd" placeholder = "password"> <br />  
  <input type = "submit" value = "submit">  
</form>
```



user name

Email

password

submit

- Things we'll implement:
  - Bind to the user's name, email, and password inputs
  - Required validation on all inputs
  - Show required validation errors
  - Disabling submit until valid
  - Submit function

# Demo

- ng g c my-form
- Import FormsModule into app.module.ts

```
import { FormsModule } from '@angular/forms';  
  
@NgModule({  
  declarations: [  
    AppComponent,  
    ContactFormComponent  
  ],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    FormsModule  
  ],  
})
```



# Demo cont : The form : Binding ngForm and ngModel

- **Step-1:** “Bind to the user’s name, email, and password inputs”.
  - What do we bind with? With ngForm and ngModel !
  - `<form novalidate #myform = "ngForm">`
  - we are exporting the ngForm value to a public #myform variable, to which we can render out the value of the form.

```
<form novalidate #myform = "ngForm">  
  <input name = "uname" placeholder = "user name"> <br/>  
  <input type = "email" name = "email" placeholder = "Email"> <br/>  
  <input type = "password" name = "passwd" placeholder = "password">  
  <input type = "submit" value = "submit">  
</form>
```

- When you imported the FormsModule in your component, Angular automatically created and attached an NgForm directive to the <form> tag in the template
- To get access to the NgForm and the overall form status, we declare a template reference variable
- The NgForm directive instance governs the form as a whole

## Demo cont : the form


- Step-2:

- <form #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >
- Next, we bind the ngSubmit event to the onSubmit() method (which is added to our component) and we pass in the form object (via the local template variable)

```
<form novalidate #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >  
  <input name = "uname" placeholder = "user name"> <br/>  
  <input type = "email" name = "email" placeholder = "Email"> <br/>  
  <input type = "password" name = "passwd" placeholder = "password">  
  <input type = "submit" value = "submit">  
</form>
```

- Test the application by clicking on "submit" button:

Shrilata	enableProdMode() to
shrillata@gmail.com	[WDS] Live Reloading
.....	form submitted
submit	>

```
my-form.component.ts X  <> my-form.component.  
c > app > my-form > TS my-form.component.ts >   
9   export class MyFormComponent {  
10  
11     constructor() { }  
12  
13     onSubmit(myform: NgForm){  
14         console.log("form submitted ");  
15     }  
16 }
```

## Demo cont : the form

- **Step-3:** Next, we register the child controls with the form. We simply add the NgModel directive and a name attribute to each element.
  - Eg : `<input type = "email" name = "email" ngModel>`
  - Is equi to : `<input type = "email" name = "email" [(ngModel)]=“email”>`

```
<form novalidate #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >
  <input name = "uname" placeholder = "user name" ngModel> <br/>
  <input type = "email" name = "email" placeholder = "Email" ngModel> <br/>
  <input type = "password" name = "passwd" placeholder = "password" ngModel>
  <input type = "submit" value = "submit">
</form>
```

```
export class MyFormComponent {
```

```
  user: User;
```

```
  constructor() { }
```

```
  onSubmit(myform: NgForm) {
```

```
    this.user = myform.value;
```

```
    console.log("form submitted " + this.user.uname);
```

```
  }
```

```
}
```

Shrilata

shrilita@gmail.com

.....

submit

enableProdMode() to enable

[WDS] Live Reloading enabl

form submitted Shrilata

>

```
export interface User{
```

```
  uname:string,
```

```
  email:string,
```

```
  passwd:string
```

```
}
```

# Demo cont

- The Template-driven forms :
  - The form is set up using ngForm directive
  - controls are set up using the ngModel directive
  - ngModel also provides the two-way data binding
  - The Validations are configured in the template via directives
- Local variables:
  - We can assign the ngForm, FormControl or FormGroup instance to a template local variable.
  - This allows us to check the status of the form like whether the form is valid, submitted, and value of the form elements, etc

# NgForm

- We have access to the ngForm instance via the local template variable #myform.

- `<form #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >`

```
<form novalidate #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >
  ...
</form>
<br>
<span>Value : {{myform.value | json }} </span><br>
<span>Valid : {{myform.valid}} </span><br>
<span>Touched : {{myform.touched }} </span><br>
<span>Submitted : {{myform.submitted }} </span><br>
```

Now, we can make use of some of the properties & methods to know the status of form. For Example

Shrilata
shrilata@gmail.com
...
submit

Value : { "uname": "Shrilata", "email": "shrilata@gmail.com", "passwd": "aaa" }

Valid : true

Touched : true

Submitted : true

user name
Email
password
submit

Value : { "uname": "", "email": "", "passwd": "" }

Valid : true

Touched : false

Submitted : false

# NgForm

- value: The value property returns the object containing the value of every FormControl
- valid: Returns true if the form is Valid else returns false.
- touched: True if the user has entered a value in at least in one field.
- submitted: Returns true if the form is submitted. else false.

```
<form novalidate #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >
```

```
  ...
```

```
</form>
```

```
<br>
```

```
<span>Value : {{myform.value | json }} </span><br>
```

```
<span>Valid : {{myform.valid}} </span><br>
```

```
<span>Touched : {{myform.touched }} </span><br>
```

```
<span>Submitted : {{myform.submitted }} </span><br>
```

Shrilata

shrilata@gmail.com

...

submit

Value : { "uname": "Shrilata", "email": "shrilata@gmail.com", "passwd": "aaa" }

Valid : true

Touched : true

Submitted : true

```
<form #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >
```

```
  <input name = "uname" placeholder = "user name" ngModel required>
```

```
  <input type = "submit" value = "submit">
```

```
</form>
```

user name

submit

Value : { "uname": "" }

Valid : false

Touched : false

Submitted : true

# FormControl

- Similarly, we can also get access to the FormControl instance by assigning the ngModel to a local variable
  - `<input name = "uname" ngModel #uname="ngModel">`
  - Now, the variable #uname holds the reference to the uname FormControl. We can then access the properties of FormControl like value, valid, invalid, touched etc

```
<input name = "uname" placeholder = "user name" ngModel #uname="ngModel">
<br>
<span>Value : {{uname.value | json }} </span><br>
<span>Valid : {{uname.valid}} </span><br>
<span>Touched : {{uname.touched }} </span><br>
<span>Invalid : {{uname.invalid }} </span><br>
```

<input type="text" value="user name"/>	<input type="text" value="Shrilata"/>
Value : ""	Value : "Shrilata"
Valid : true	Valid : true
Touched : false	Touched : true
Invalid : false	Invalid : false

- value: Returns the current value of the control
- valid: Returns true if the value is Valid else false
- invalid: True if the value is invalid else false
- touched: Returns true if the value is entered in the element

# Template driven form validation

- It is very common that the users will make mistakes when filling out the web form.
- This is where the validations come into play.
- The validation module must ensure that the user has provided accurate and complete information in the form fields.
- We must display the validation error messages to the users, disable the submit button until validation.
- Lets see how to use the Angular built-in validators.
  - Validations in Template-driven forms are provided by the Validation directives.
  - The Angular Forms Module comes with several built-in validators.
  - You can also create your own custom Validator.



# Template-driven error validation

- Step-1 :

- First, we need to disable browser validator interfering with the Angular validator.
- To do that we need to add novalidate attribute on <form> element
- `<form novalidate #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >`
- novalidate: This will prevent the default HTML5 validations since we'll be doing that ourselves

- Step-2 :

- You will notice that the click submit button still submits the form.
- We need to disable the submit button if our form is not valid.
- `<input type = "submit" value = "submit" [disabled]="!myform.valid">`
- So long as myform.valid remains false, the submit button remains disabled.

```
<form    novalidate #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >
  ...
  <input type = "submit" value = "submit" [disabled]="!myform.valid">
</form>
```

# Adding in Built-in Validators

- **Required Validation** : returns true only if the form control has non-empty value entered. Let us add this validator to all fields

```
<input name="uname" required placeholder="user name" ngModel #uname="ngModel">  
<input type="email" required name="email" placeholder="Email" ngModel> <br />  
<input type="password" required name="passwd" placeholder="password" ngModel>
```

- **Minlength Validation** : requires the control value must not have less number of characters than the value specified in the validator.
  - For Example, minlength validator ensures that the username value has at least 10 characters.
- **Maxlength Validation** : requires that the number of characters must not exceed the value of the attribute.
  - Eg : `<input name="uname" required minlength="6" maxlength="15">`

# Adding in Built-in Validators

- **Pattern Validation** : requires that the control value must match the regex pattern provided in the attribute.
  - For example, the pattern `^[a-zA-Z]+$` ensures that the only letters are allowed (even spaces are not allowed).
- **Email Validation** : requires that the control value must be a valid email address. We apply this to the email field

```
<input name="uname"
      required
      minlength="6"
      maxlength="15"
      pattern="^[a-zA-Z]+$"
      placeholder="user name"
      ngModel #uname="ngModel">
<input type="email"
      required
      name="email"
      placeholder="Email"
      ngModel> <br />
```

# Displaying the Validation/Error messages

- We need to provide a short and meaningful error message to the user.
  - Angular creates a FormControl for each and every field, which has ngModel directive applied.
  - The FormControl exposes the state of form element like valid, dirty, touched, etc.
  - There are two ways in which you can get the reference to the FormControl.
    - One way is to use the myform variable. We can use the myform.controls.uname.valid to find out if the uname is valid.
    - The other way is to create a new local variable for each FormControl
    - For Example, the following #uname="ngModel" creates the uname variable with the FormControl instance.

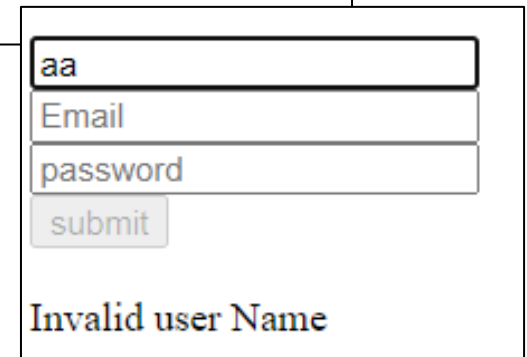
```
<input name="uname"  
      required  
      ngModel  
      #uname="ngModel"
```

# Displaying the Validation/Error messages

- Now, we have a reference to the username FormControl instance, we can check its status.
- We use the valid property to check if the username has any errors.
- valid: returns either invalid status or null which means a valid status
- Now, we have a reference to the username FormControl instance, we can check its status.

```
<input name="username"
      required
      ngModel
      #username="ngModel"
      minlength="6"
      maxlength="15"
      pattern="^[a-zA-Z]+$"
      placeholder="user name">
```

```
<div *ngIf="!username.valid && (username.dirty || username.touched)">
  Invalid user Name
</div>
```



aa

Email

password

submit

Invalid user Name

# Displaying the Validation/Error messages

- Why check dirty and touched?
  - We do not want the application to display the error when the form is displayed for the first time.
  - We want to display errors only after the user has attempted to change the value. The dirty & touched properties help us do that.
- dirty: A control is dirty if the user has changed the value in the UI.
- touched: A control is touched if the user has triggered a blur event on it.

# Error message

- The error message "Invalid First Name" is not helpful. The firstname has two validators. required and minlength
  - Any errors generated by the failing validation is updated in the errors object.
  - The errors object returns the error object or null if there are no errors.

```
<input name="uname"
  required
  ngModel
  #uname="ngModel"
  minlength="6"
  maxlength="15"
  pattern="^[a-zA-Z]+$"
  placeholder="user name"
```

```
> <br />
```

```
<div *ngIf="!uname?.valid && (uname?.dirty || uname?.touched)">
```

```
  Invalid User Name
```

```
  <div *ngIf="uname.errors.required">
```

```
    User Name is required
```

```
  </div>
```

```
  <div *ngIf="uname.errors.minlength">
```

```
    User Name Minimum Length is {{uname.errors.minlength?.requiredLength}}
```

```
  </div>
```

```
</div>
```

Invalid User Name

User Name Minimum Length is 6