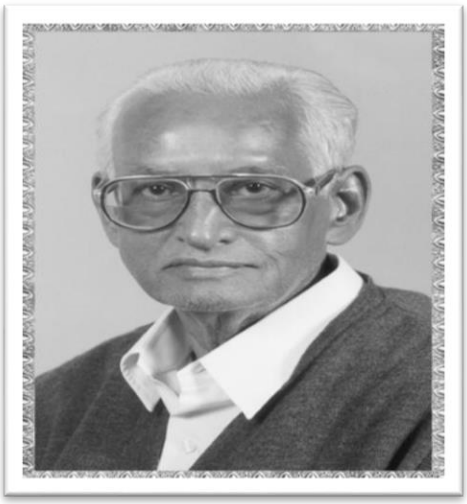
SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.
(AFFILIATED TO SAURASHTRA UNIVERSITY)



Lt. Shree Chimanbhai Shukla

MSCIT SEM-3 ANGULARJS

Shree H.N.Shukla college2
vaishali nagar
Near Amrapali Under Bridge,
Raiya road
Rajkot
Ph No:-0281 2440478

Shree H.N.Shukla college3
vaishali nagar
Near Amrapali Under Bridge,
Raiya road
Rajkot
Ph No:-0281 2440478

Unit : 5

Form Handling and Even Handling

- **Introduction to Form Handling**
- **Form Validation**
- **ng-minlength**
- **ng-required**
- **ng-pattern**
- **ngmaxlength**
- **ng-minlength**
- **Form Validation**
- **Submitting Forms**
- **Event Handling with Forms**

Introduction to forms in Angular

Handling user input with forms is the cornerstone of many common applications. Applications use forms to enable users to log in, to update a profile, to enter sensitive information, and to perform many other data-entry tasks.

Angular provides two different approaches to handling user input through forms: reactive and template-driven. Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.

This guide provides information to help you decide which type of form works best for your situation. It introduces the common building blocks used by both approaches. It also summarizes the key differences between the two approaches, and demonstrates those differences in the context of setup, data flow, and testing.

❖ Prerequisites

This guide assumes that you have a basic understanding of the following.

- TypeScript and HTML5 programming
- Angular app-design fundamentals, as described in Angular Concepts
- The basics of Angular template syntax

Choosing an approach

Reactive forms and template-driven forms process and manage form data differently. Each approach offers different advantages.

FORMS	DETAILS
Reactive forms	Provide direct, explicit access to the underlying form's object model. Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.
Template-driven forms	Rely on directives in the template to create and manipulate the underlying object model. They are useful for adding a simple form to an app, such as an email list signup form. They're straightforward to add to an app, but they don't scale as well as reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, template-driven forms could be a good fit.

Key differences

The following table summarizes the key differences between reactive and template-driven forms.

	REACTIVE	TEMPLATE-DRIVEN
Setup of form model	Explicit, created in component class	Implicit, created by directives
Data model	Structured and immutable	Unstructured and mutable
Data flow	Synchronous	Asynchronous
Form validation	Functions	Directives

❖ Scalability

If forms are a central part of your application, scalability is very important. Being able to reuse form models across components is critical.

Reactive forms are more scalable than template-driven forms. They provide direct access to the underlying form API, and use synchronous data flow between the view and the data model, which makes creating large-scale forms easier. Reactive forms require less setup for testing, and testing does not require deep understanding of change detection to properly test form updates and validation.

Template-driven forms focus on simple scenarios and are not as reusable. They abstract away the underlying form API, and use asynchronous data flow between the view and the data model. The abstraction of template-driven forms also affects testing. Tests are deeply reliant on manual change detection execution to run properly, and require more setup.

Setting up the form model

Both reactive and template-driven forms track value changes between the form input elements that users interact with and the form data in your component model. The two approaches share underlying building blocks, but differ in how you create and manage the common form-control instances.

Common form foundation classes

Both reactive and template-driven forms are built on the following base classes.

BASE CLASSES	DETAILS

BASE CLASSES	DETAILS
FormControl	Tracks the value and validation status of an individual form control.
FormGroup	Tracks the same values and status for a collection of form controls.
FormArray	Tracks the same values and status for an array of form controls.
ControlValueAccessor	Creates a bridge between Angular FormControl instances and built-in DOM elements.

❖ Setup in reactive forms

With reactive forms, you define the form model directly in the component class. The [formControl] directive links the explicitly created FormControl instance to a specific form element in the view, using an internal value accessor.

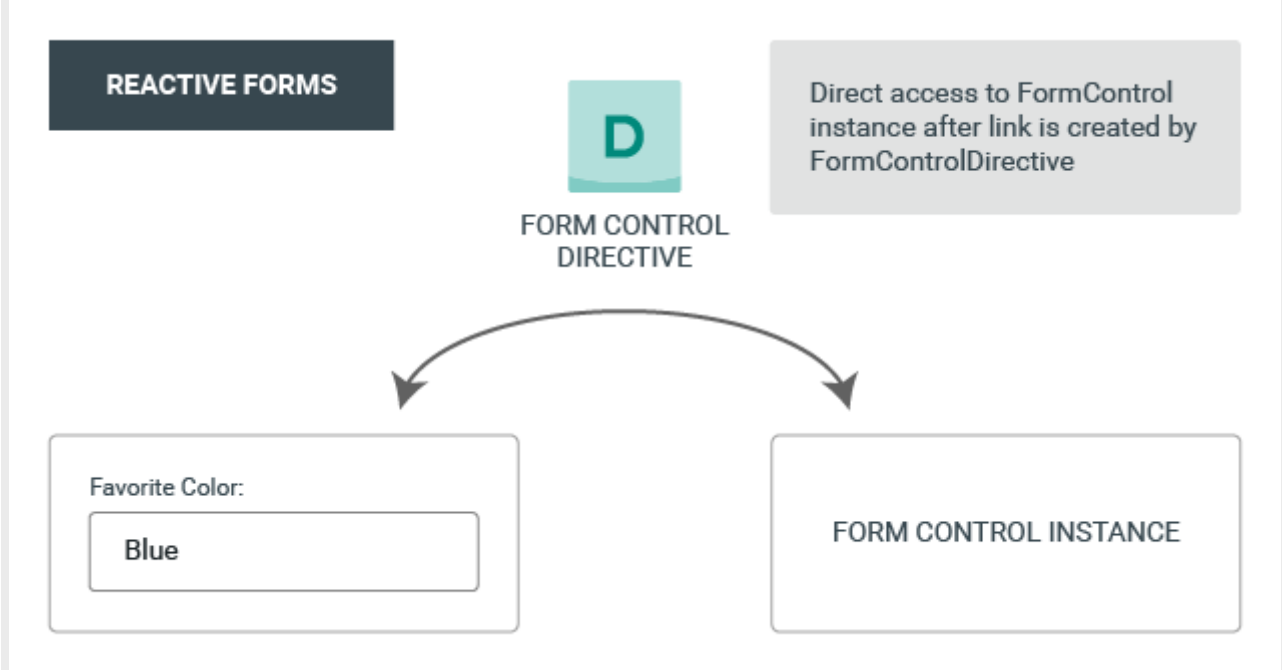
The following component implements an input field for a single control, using reactive forms. In this example, the form model is the FormControl instance.

```
content_copyimport { Component } from '@angular/core';

import { FormControl } from '@angular/forms';


@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text" [formControl]="favoriteColorControl">
  `
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
```

```
}
```

Figure 1 shows how, in reactive forms, the form model is the source of truth; it provides the value and status of the form element at any given point in time, through the [formControl] directive on the input element.

Figure 1. Direct access to forms model in a reactive form.



Setup in template-driven forms

In template-driven forms, the form model is implicit, rather than explicit. The directive NgModel creates and manages a FormControl instance for a given form element.

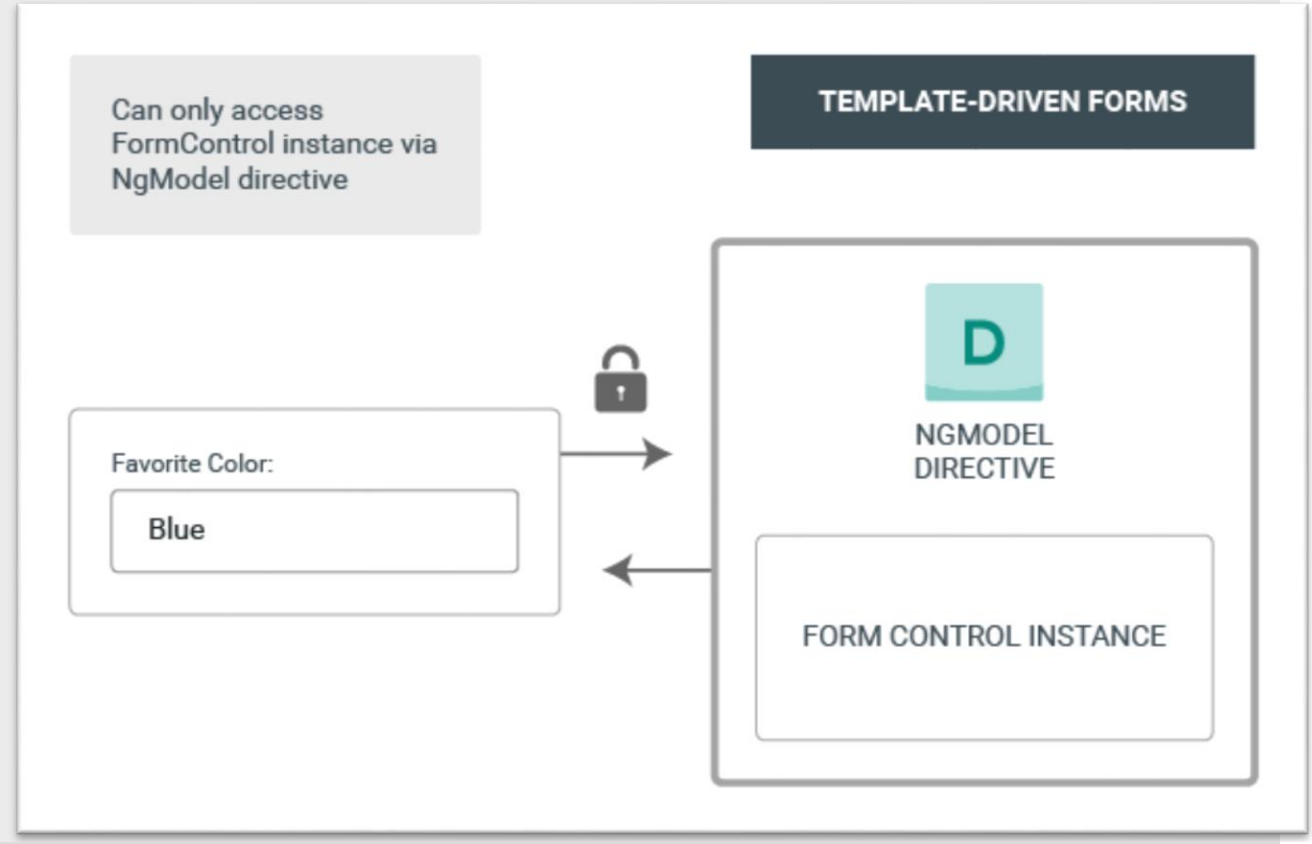
The following component implements the same input field for a single control, using template-driven forms.

```
content_copyimport { Component } from '@angular/core';


@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
```

```
  `
})
export class FavoriteColorComponent {
  favoriteColor = '';
}
```

In a template-driven form the source of truth is the template. You do not have direct programmatic access to the FormControl instance, as shown in Figure 2.

Figure 2. Indirect access to forms model in a template-driven form.



❖ Data flow in forms

When an application contains a form, Angular must keep the view in sync with the component model and the component model in sync with the view. As users change values and make

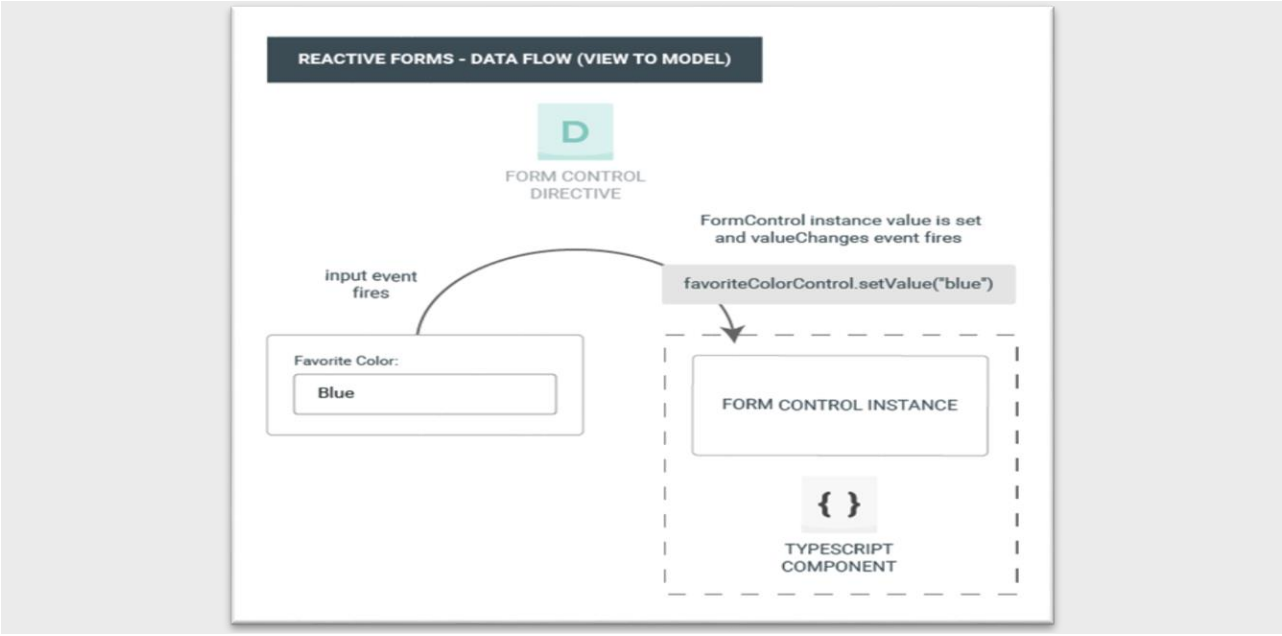
selections through the view, the new values must be reflected in the data model. Similarly, when the program logic changes values in the data model, those values must be reflected in the view.

Reactive and template-driven forms differ in how they handle data flowing from the user or from programmatic changes. The following diagrams illustrate both kinds of data flow for each type of form, using the favorite-color input field defined above.
Data flow in reactive forms

In reactive forms each form element in the view is directly linked to the form model (a FormControl instance). Updates from the view to the model and from the model to the view are synchronous and do not depend on how the UI is rendered.

The view-to-model diagram shows how data flows when an input field's value is changed from the view through the following steps.

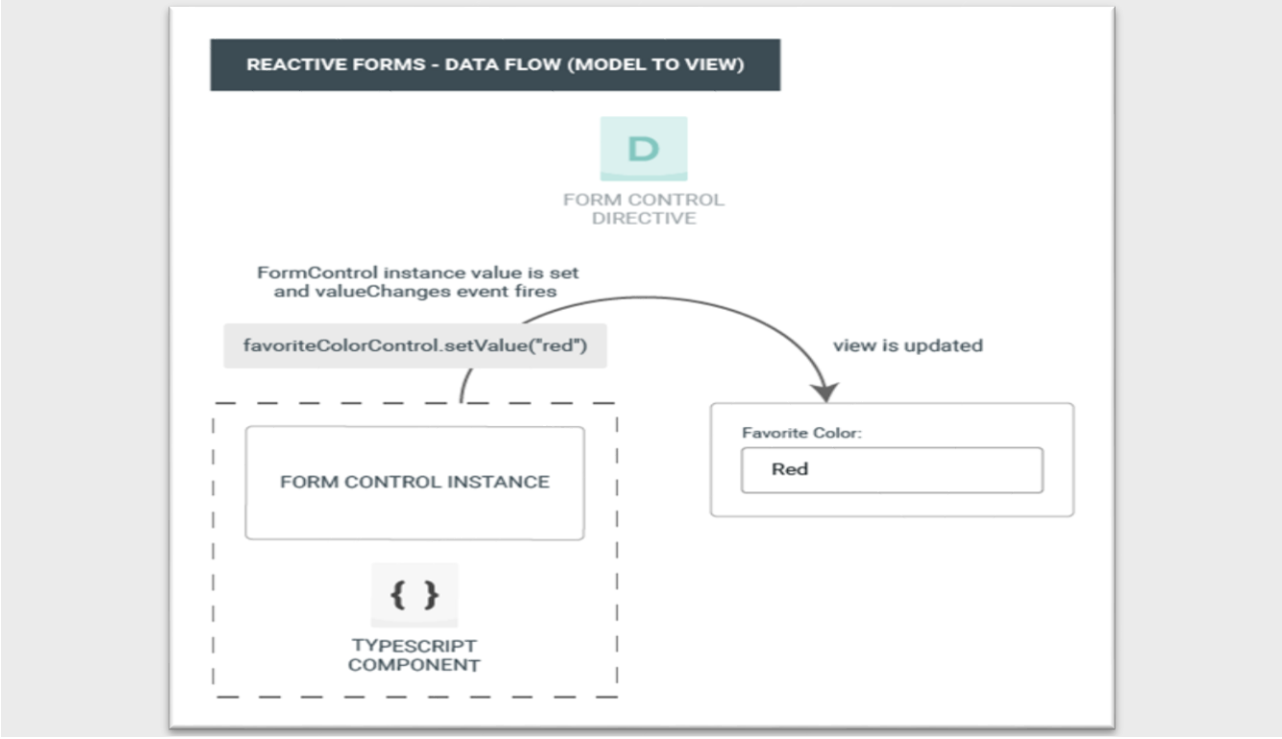
1. The user types a value into the input element, in this case the favorite color *Blue*.
2. The form input element emits an "input" event with the latest value.
3. The control value accessor listening for events on the form input element immediately relays the new value to the FormControl instance.
4. The FormControl instance emits the new value through the valueChanges observable.
5. Any subscribers to the valueChanges observable receive the new value.



The model-to-view diagram shows how a programmatic change to the model is propagated to the view through the following steps.

1. The user calls the favoriteColorControl.setValue() method, which updates the FormControl value.

2. The FormControl instance emits the new value through the valueChanges observable.
3. Any subscribers to the valueChanges observable receive the new value.
4. The control value accessor on the form input element updates the element with the new value.

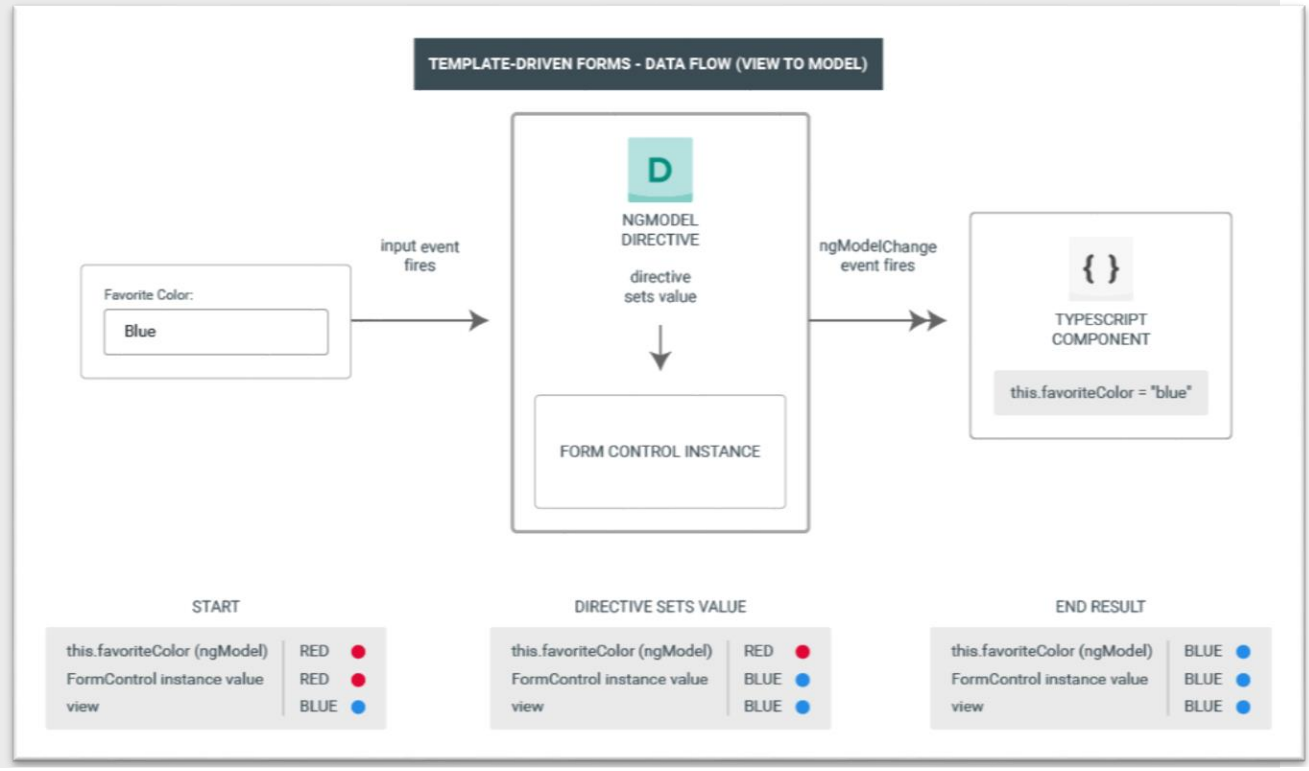


❖ Data flow in template-driven forms

In template-driven forms, each form element is linked to a directive that manages the form model internally.

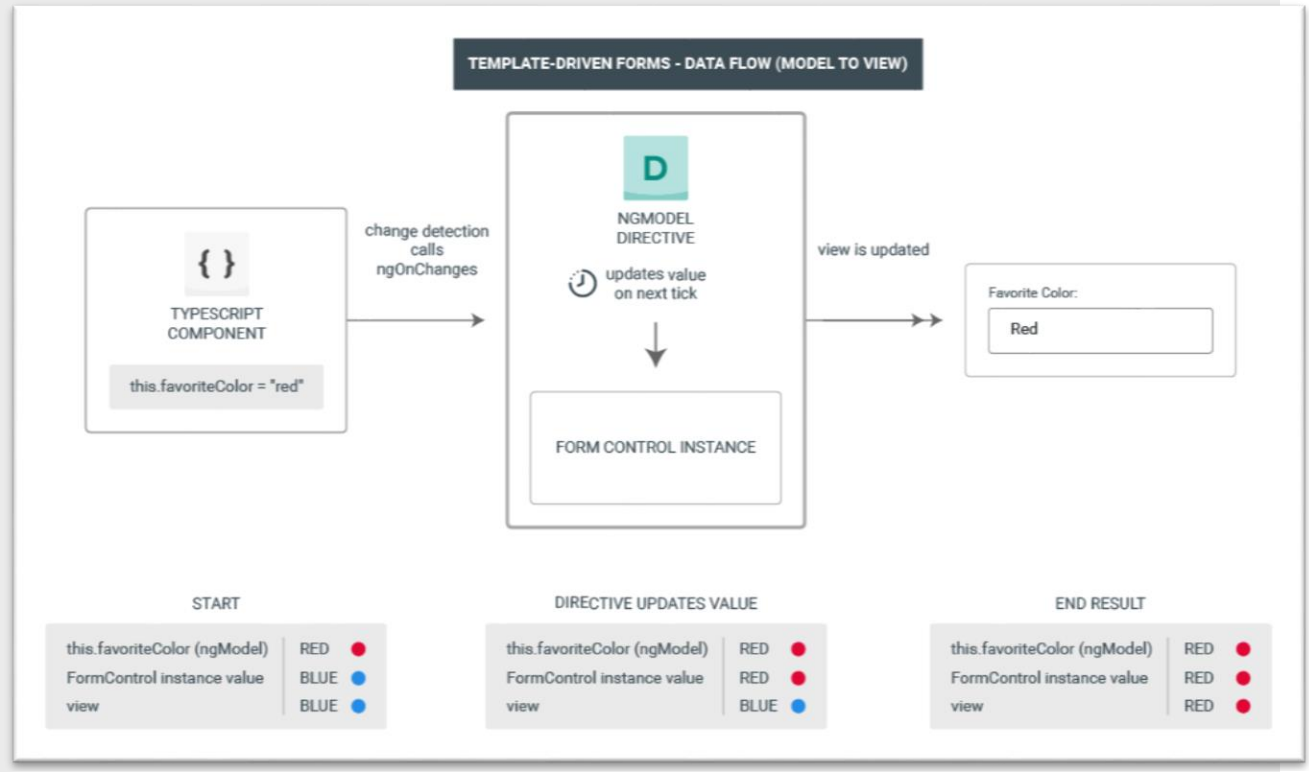
The view-to-model diagram shows how data flows when an input field's value is changed from the view through the following steps.

1. The user types *Blue* into the input element.
2. The input element emits an "input" event with the value *Blue*.
3. The control value accessor attached to the input triggers the setValue() method on the FormControl instance.
4. The FormControl instance emits the new value through the valueChanges observable.
5. Any subscribers to the valueChanges observable receive the new value.
6. The control value accessor also calls the NgModel.viewToModelUpdate() method which emits an ngModelChange event.
7. Because the component template uses two-way data binding for the favoriteColor property, the favoriteColor property in the component is updated to the value emitted by the ngModelChange event (*Blue*).



The model-to-view diagram shows how data flows from model to view when the favoriteColor changes from *Blue* to *Red*, through the following steps

1. The favoriteColor value is updated in the component.
2. Change detection begins.
3. During change detection, the ngOnChanges lifecycle hook is called on the NgModel directive instance because the value of one of its inputs has changed.
4. The ngOnChanges() method queues an async task to set the value for the internal FormControl instance.
5. Change detection completes.
6. On the next tick, the task to set the FormControl instance value is executed.
7. The FormControl instance emits the latest value through the valueChanges observable.
8. Any subscribers to the valueChanges observable receive the new value.
9. The control value accessor updates the form input element in the view with the latest favoriteColor value.



❖ **Mutability of the data model**

The change-tracking method plays a role in the efficiency of your application.

FORMS	DETAILS
Reactive forms	Keep the data model pure by providing it as an immutable data structure. Each time a change is triggered on the data model, the FormControl instance returns a new data model rather than updating the existing data model. This gives you the ability to track unique changes to the data model through the control's observable. Change detection is more efficient because it only needs to update on unique changes. Because data updates follow reactive patterns, you can integrate with observable operators to transform data.
Template-driven forms	Rely on mutability with two-way data binding to update the data model in the component as changes are made in the template. Because there are no unique changes to track on the data model when using two-way data binding, change detection is less efficient at determining when updates are required.

The difference is demonstrated in the previous examples that use the favorite-color input element.

- With reactive forms, the FormControl instance always returns a new value when the control's value is updated
- With template-driven forms, the favorite color property is always modified to its new value

Form validation

Validation is an integral part of managing any set of forms. Whether you're checking for required fields or querying an external API for an existing username, Angular provides a set of built-in validators as well as the ability to create custom validators.

FORMS	DETAILS
Reactive forms	Define custom validators as functions that receive a control to validate
Template-driven forms	Tied to template directives, and must provide custom validator directives that wrap validation functions

For more information, see Form Validation.

❖ Testing

Testing plays a large part in complex applications. A simpler testing strategy is useful when validating that your forms function correctly. Reactive forms and template-driven forms have different levels of reliance on rendering the UI to perform assertions based on form control and form field changes. The following examples demonstrate the process of testing forms with reactive and template-driven forms.

❖ Testing reactive forms

Reactive forms provide a relatively straightforward testing strategy because they provide synchronous access to the form and data models, and they can be tested without rendering the UI. In these tests, status and data are queried and manipulated through the control without interacting with the change detection cycle.

The following tests use the favorite-color components from previous examples to verify the view-to-model and model-to-view data flows for a reactive form.

❖ Verifying view-to-model data flow

The first example performs the following steps to verify the view-to-model data flow.

1. Query the view for the form input element, and create a custom "input" event for the test.

2. Set the new value for the input to *Red*, and dispatch the "input" event on the form input element.
3. Assert that the component's favoriteColorControl value matches the value from the input.

Favorite color test - view to model

```
content_copyit('should update the value of the input field', () => {
  const input = fixture.nativeElement.querySelector('input');
  const event = createNewEvent('input');

  input.value = 'Red';
  input.dispatchEvent(event);

  expect(fixture.componentInstance.favoriteColorControl.value).toEqual('Red');
});
```

The next example performs the following steps to verify the model-to-view data flow.

1. Use the favoriteColorControl, a FormControl instance, to set the new value.
2. Query the view for the form input element.
3. Assert that the new value set on the control matches the value in the input.

Favorite color test - model to view

```
content_copyit('should update the value in the control', () => {
  component.favoriteColorControl.setValue('Blue');

  const input = fixture.nativeElement.querySelector('input');

  expect(input.value).toBe('Blue');
});
```

❖ Testing template-driven forms

Writing tests with template-driven forms requires a detailed knowledge of the change detection process and an understanding of how directives run on each cycle to ensure that elements are queried, tested, or changed at the correct time.

The following tests use the favorite color components mentioned earlier to verify the data flows from view to model and model to view for a template-driven form.

The following test verifies the data flow from view to model.

❖ Favorite color test - view to model

```
content_copyit('should update the favorite color in the component', fakeAsync(() => {
    const input = fixture.nativeElement.querySelector('input');
    const event = createNewEvent('input');
    input.value = 'Red';
```

```
    input.dispatchEvent(event);

    fixture.detectChanges();

    expect(component.favoriteColor).toEqual('Red');
  }));
```

Here are the steps performed in the view to model test.

1. Query the view for the form input element, and create a custom "input" event for the test.
2. Set the new value for the input to *Red*, and dispatch the "input" event on the form input element.
3. Run change detection through the test fixture.
4. Assert that the component favoriteColor property value matches the value from the input.

The following test verifies the data flow from model to view.

Favorite color test - model to view

```
content_copyit('should update the favorite color on the input field', fakeAsync(() => {

    component.favoriteColor = 'Blue';

    fixture.detectChanges();

    tick();

    const input = fixture.nativeElement.querySelector('input');

    expect(input.value).toBe('Blue');  }));
```

Here are the steps performed in the model to view test.

1. Use the component instance to set the value of the favoriteColor property.
2. Run change detection through the test fixture.
3. Use the tick() method to simulate the passage of time within the fakeAsync() task.
4. Query the view for the form input element.
5. Assert that the input value matches the value of the favoriteColor property in the component instance.

ngMinlength

ngMinlength adds the minlength validator to ngModel. It is most often used for text-based input controls, but can also be applied to custom text-based controls.
The validator sets the minlength error key if the ngModel.$viewValue is shorter than the integer obtained by evaluating the AngularJS expression given in the ngMinlength attribute value.
Note: This directive is also added when the plain minlength attribute is used, with two differences: ngMinlength does not set the minlength attribute and therefore HTML5 constraint validation is not available.
The ngMinlength value must be an expression, while the minlength value must be interpolated.

Directive Info

- This directive executes at priority level 0.

Usage

- as attribute:

```
<ANY
  ng-minlength="expression">
...
</ANY>
```

Arguments

Param	Type	Details
ngMinlength	expression	AngularJS expression that must evaluate to a Number or String parsable into a Number. Used as value for the minlength validator.

Example

Edit in Plunker

index.htmlprotractor.js

```
<script>
  angular.module('ngMinlengthExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
```

```
    $scope.minlength = 3;
  }]);
</script>
<div ng-controller="ExampleController">
 <form name="form">
  <label for="minlength">Set a minlength: </label>
  <input type="number" ng-model="minlength" id="minlength" />
  <br>
  <label for="input">This input is restricted by the current minlength: </label>
  <input type="text" ng-model="model" id="input" name="input" ng-minlength="minlength" /><br>
  <hr>
  input valid? = <code>{{form.input.$valid}}</code><br>
  model = <code>{{model}}</code>
 </form>
</div>
```

ngMaxlength

Overview

ngMaxlength adds the maxlength validator to ngModel. It is most often used for text-based input controls, but can also be applied to custom text-based controls.

The validator sets the maxlength error key if the ngModel.$viewValue is longer than the integer obtained by evaluating the AngularJS expression given in the ngMaxlength attribute value.

Note: This directive is also added when the plain maxlength attribute is used, with two differences:

1. ngMaxlength does not set the maxlength attribute and therefore HTML5 constraint validation is not available.
2. The ngMaxlength attribute must be an expression, while the maxlength value must be interpolated.

Directive Info

- This directive executes at priority level 0.

Usage

as attribute:

- <ANY

```
- ng-maxlength="expression">.
</ANY>
```

Arguments

Param	Type	Details
ngMaxlength	expression	AngularJS expression that must evaluate to a Number or String parsable into a Number. Used as value for the maxlength validator.

Example

Edit in Plunker

index.htmlprotractor.js

```
<script>
  angular.module('ngMaxlengthExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.maxlength = 5;
    }]);
</script>
<div ng-controller="ExampleController">
  <form name="form">
    <label for="maxlength">Set a maxlength: </label>
    <input type="number" ng-model="maxlength" id="maxlength" />
    <br>
    <label for="input">This input is restricted by the current maxlength: </label>
    <input type="text" ng-model="model" id="input" name="input" ng-maxlength="maxlength" /><br>
    <hr>
    input valid? = <code>{{form.input.$valid}}</code><br>
    model = <code>{{model}}</code>
  </form>
</div>
```

ngPattern

Overview

ngPattern adds the pattern validator to ngModel. It is most often used for text-based input controls, but can also be applied to custom text-based controls.
The validator sets the pattern error key if the ngModel.$viewValue does not match a RegExp which is obtained from the ngPattern attribute value:

- the value is an AngularJS expression:
  - If the expression evaluates to a RegExp object, then this is used directly.
  - If the expression evaluates to a string, then it will be converted to a RegExp after wrapping it in ^ and $ characters. For instance, "abc" will be converted to new RegExp('^abc$').
- If the value is a RegExp literal, e.g. ngPattern="/^\d+$/", it is used directly.

Note: Avoid using the g flag on the RegExp, as it will cause each successive search to start at the index of the last search's match, thus not taking the whole input value into account.
Note: This directive is also added when the plain pattern attribute is used, with two differences:

1. ngPattern does not set the pattern attribute and therefore HTML5 constraint validation is not available.
2. The ngPattern attribute must be an expression, while the pattern value must be interpolated.

Directive Info

- This directive executes at priority level 0.

Usage

- as attribute:

```
<ANY
  ng-pattern="">
...
</ANY>
```

Arguments

Param	Type	Details

Param	Type	Details
ngPattern	expressionRegExp	AngularJS expression that must evaluate to a `RegExp` or a `String` parsable into a `RegExp`, or a `RegExp` literal. See above for more details.

Example

Edit in Plunker

index.htmlprotractor.js

```
<script>
  angular.module('ngPatternExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.regex = '\\d+';
    }]);
</script>
<div ng-controller="ExampleController">
  <form name="form">
    <label for="regex">Set a pattern (regex string): </label>
    <input type="text" ng-model="regex" id="regex" />
    <br>
    <label for="input">This input is restricted by the current pattern: </label>
    <input type="text" ng-model="model" id="input" name="input" ng-pattern="regex" /><br>
    <hr>
     input valid? = <code>{{form.input.$valid}}</code><br>
    model = <code>{{model}}</code>
  </form>
</div>
```

ngRequired

Overview

ngRequired adds the required validator to ngModel. It is most often used for input and select controls, but can also be applied to custom controls.
The directive sets the required attribute on the element if the AngularJS expression inside ngRequired evaluates to true. A special directive for setting required is necessary because we cannot use interpolation inside required. See the interpolation guide for more info.
The validator will set the required error key to true if the required attribute is set and calling NgModelController.$isEmpty with the ngModel.$viewValue returns true. For example, the $isEmpty() implementation for input[text] checks the length of the $viewValue. When developing custom controls, $isEmpty() can be overwritten to account for a $viewValue that is not string-based.

Directive Info

- This directive executes at priority level 0.

Usage

- as attribute:

```
<ANY
  ng-required="expression">
...
</ANY>
```

Arguments

Param	Type	Details
ngRequired	expression	AngularJS expression. If it evaluates to `true`, it sets the `required` attribute to the element and adds the `required` `validator`.

Example

Edit in Plunker

index.htmlprotractor.js

```
<script>
  angular.module('ngRequiredExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.required = true;
    }]);
</script>
<div ng-controller="ExampleController">
  <form name="form">
    <label for="required">Toggle required: </label>
    <input type="checkbox" ng-model="required" id="required" />
    <br>
    <label for="input">This input must be filled if `required` is true: </label>
    <input type="text" ng-model="model" id="input" name="input" ng-required="required" /><br>
    <hr>
    required error set? = <code>{{form.input.$error.required}}</code><br>
    model = <code>{{model}}</code>
  </form>
</div>
```

ngSubmit

Overview

Enables binding AngularJS expressions to onsubmit events.
Additionally it prevents the default action (which for form means sending the request to the server and reloading the current page), but only if the form does not contain action, data-action, or x-action attributes.

Warning: Be careful not to cause "double-submission" by using both the ngClick and ngSubmit handlers together. See the form directive documentation for a detailed discussion of when ngSubmit may be triggered.

Directive Info

- This directive executes at priority level 0.

Usage

- as attribute:

```
<form
```

```
    ng-submit="expression">
  ...
</form>
```

Arguments

Param	Type	Details
ngSubmit	expression	Expression to eval. (Event object is available as `$event`)

Example

Edit in Plunker

index.htmlprotractor.js

```
<script>
  angular.module('submitExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.list = [];
      $scope.text = 'hello';
      $scope.submit = function() {
        if ($scope.text) {
          $scope.list.push(this.text);
          $scope.text = '';
        }
      };
    }]);
</script>
<form ng-submit="submit()" ng-controller="ExampleController">
  Enter text and hit enter:
  <input type="text" ng-model="text" name="text" />
  <input type="submit" id="submit" value="Submit" />
  <pre>list={{list}}</pre>
</form>
```

AngularJS - Forms

AngularJS enriches form filling and validation. We can use ng-click event to handle the click button and use $dirty and $invalid flags to do the validation in a seamless way. Use novalidate with a form declaration to disable any browser-specific validation. The form controls make heavy use of AngularJS events. Let us have a look at the events first.

❖ Events

AngularJS provides multiple events associated with the HTML controls. For example, ng-click directive is generally associated with a button. AngularJS supports the following events –

- **ng-click**
- **ng-dbl-click**
- **ng-mousedown**
- **ng-mouseup**
- **ng-mouseenter**
- **ng-mouseleave**
- **ng-mousemove**
- **ng-mouseover**
- **ng-keydown**
- **ng-keyup**
- **ng-keypress**
- **ng-change**

❖ ng-click

Reset data of a form using on-click directive of a button.

```
<input name = "firstname" type = "text" ng-model = "firstName" required>
<input name = "lastname" type = "text" ng-model = "lastName" required>
<input name = "email" type = "email" ng-model = "email" required>
<button ng-click = "reset()">Reset</button>

<script>
  function studentController($scope) {
    $scope.reset = function() {
      $scope.firstName = "Mahesh";
      $scope.lastName = "Parashar";
      $scope.email = "MaheshParashar@tutorialspoint.com";
    }

    $scope.reset();
  }
```

```
</script>
```

Validate Data

The following can be used to track error.

- **$dirty** − states that value has been changed.
- **$invalid** − states that value entered is invalid.
- **$error** − states the exact error.

Example

The following example will showcase all the above-mentioned directives.

testAngularJS.htm
Live Demo

```
<html>
  <head>
    <title>Angular JS Forms</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

    <style>
      table, th , td {
        border: 1px solid grey;
        border-collapse: collapse;
        padding: 5px;
      }
      table tr:nth-child(odd) {
        background-color: #f2f2f2;
      }
      table tr:nth-child(even) {
        background-color: #ffffff;
      }
    </style>

  </head>
  <body>

    <h2>AngularJS Sample Application</h2>
```

```
    <div ng-app = "mainApp" ng-controller = "studentController">

      <form name = "studentForm" novalidate>
        <table border = "0">
          <tr>
            <td>Enter first name:</td>
            <td><input name = "firstname" type = "text" ng-model = "firstName"
required>
              <span style = "color:red" ng-show = "studentForm.firstname.$dirty &&
studentForm.firstname.$invalid">
                <span ng-show = "studentForm.firstname.$error.required">First Name is
required.</span>
              </span>
            </td>
          </tr>

          <tr>
            <td>Enter last name: </td>
            <td><input name = "lastname"  type = "text" ng-model = "lastName"
required>
              <span style = "color:red" ng-show = "studentForm.lastname.$dirty &&
studentForm.lastname.$invalid">
                <span ng-show = "studentForm.lastname.$error.required">Last Name is
required.</span>
              </span>
            </td>
          </tr>

          <tr>
            <td>Email: </td><td><input name = "email" type = "email" ng-model =
"email" length = "100" required>
              <span style = "color:red" ng-show = "studentForm.email.$dirty &&
studentForm.email.$invalid">
                <span ng-show = "studentForm.email.$error.required">Email is
required.</span>
                <span ng-show = "studentForm.email.$error.email">Invalid email
address.</span>
              </span>
            </td>
```

```
            </tr>

            <tr>
              <td>
                <button ng-click = "reset()">Reset</button>
              </td>
              <td>
                <button ng-disabled = "studentForm.firstname.$dirty &&
                  studentForm.firstname.$invalid || studentForm.lastname.$dirty &&
                  studentForm.lastname.$invalid || studentForm.email.$dirty &&
                  studentForm.email.$invalid" ng-click="submit()">Submit</button>
              </td>
            </tr>

          </table>
        </form>
      </div>

      <script>
        var mainApp = angular.module("mainApp", []);

        mainApp.controller('studentController', function($scope) {
          $scope.reset = function() {
            $scope.firstName = "Mahesh";
            $scope.lastName = "Parashar";
            $scope.email = "MaheshParashar@tutorialspoint.com";
          }

          $scope.reset();
        });
      </script>

    </body>
</html>
```


Output
Open the file testAngularJS.htm in a web browser and see the result.
AngularJS Sample Application
Enter first name:
Mahesh
Enter last name:
Parashar
Email:
MaheshParashar@tutorialsp