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.

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

In general:

- **Reactive forms** 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** are useful for adding a simple form to an app, such as an email list signup form. They're easy 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, use template-driven forms.

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.

> **Note:** For complete information about each kind of form, see Reactive Forms and Template-driven Forms.

Key differences

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

	REACTIVE	TEMPLATE-DRIVEN
Setup (form model)	More explicit, created in component class	Less explicit, created by directives
Data model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Form validation	Functions	Directives
Mutability	Immutable	Mutable
Scalability	Low-level API access	Abstraction on top of APIs

Common foundation

Both reactive and template-driven forms share underlying building blocks.

- `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 native DOM elements.

See the Form model setup section below for an introduction to how these control instances are created and managed with reactive and template-driven forms. Further details are provided in the data flow section of this guide.

Form model setup

Reactive and template-driven forms both use a form model to track value changes between Angular forms and form input elements. The examples below show how the form model is defined and created.

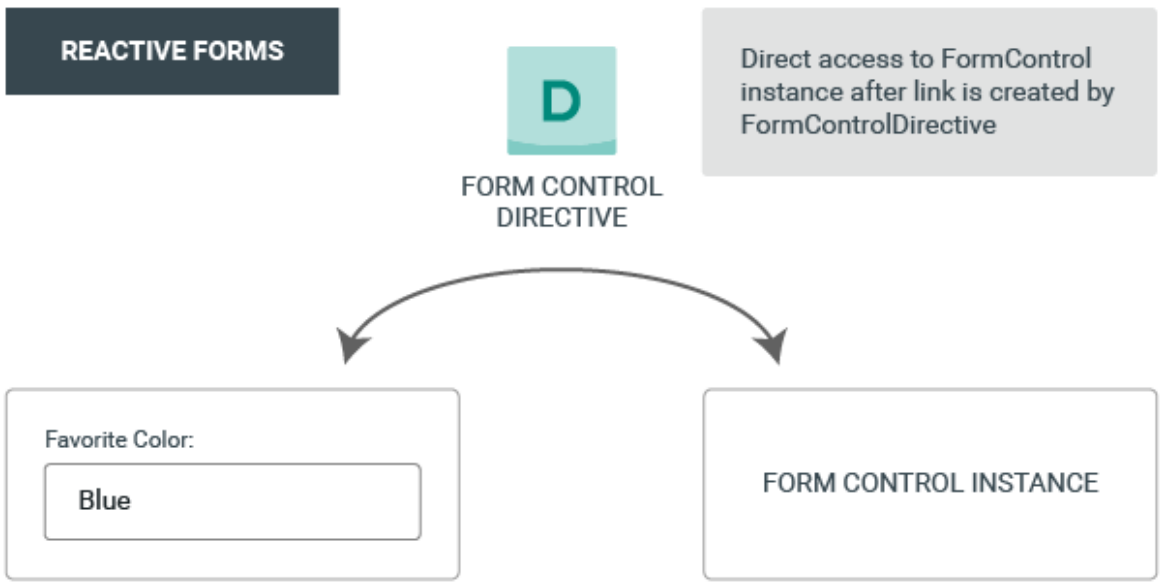
Setup in reactive forms

Here's a component with an input field for a single control implemented using reactive forms.

```
import { 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('');
}
```

The source of truth provides the value and status of the form element at a given point in time. In reactive forms, the form model is the source of truth. In the example above, the form model is the `FormControl` instance.



With reactive forms, the form model is explicitly defined in the component class. The reactive form directive (in this case, `FormControlDirective`) then links the existing `FormControl` instance to a specific form element in the view using a value accessor (`ControlValueAccessor` instance).

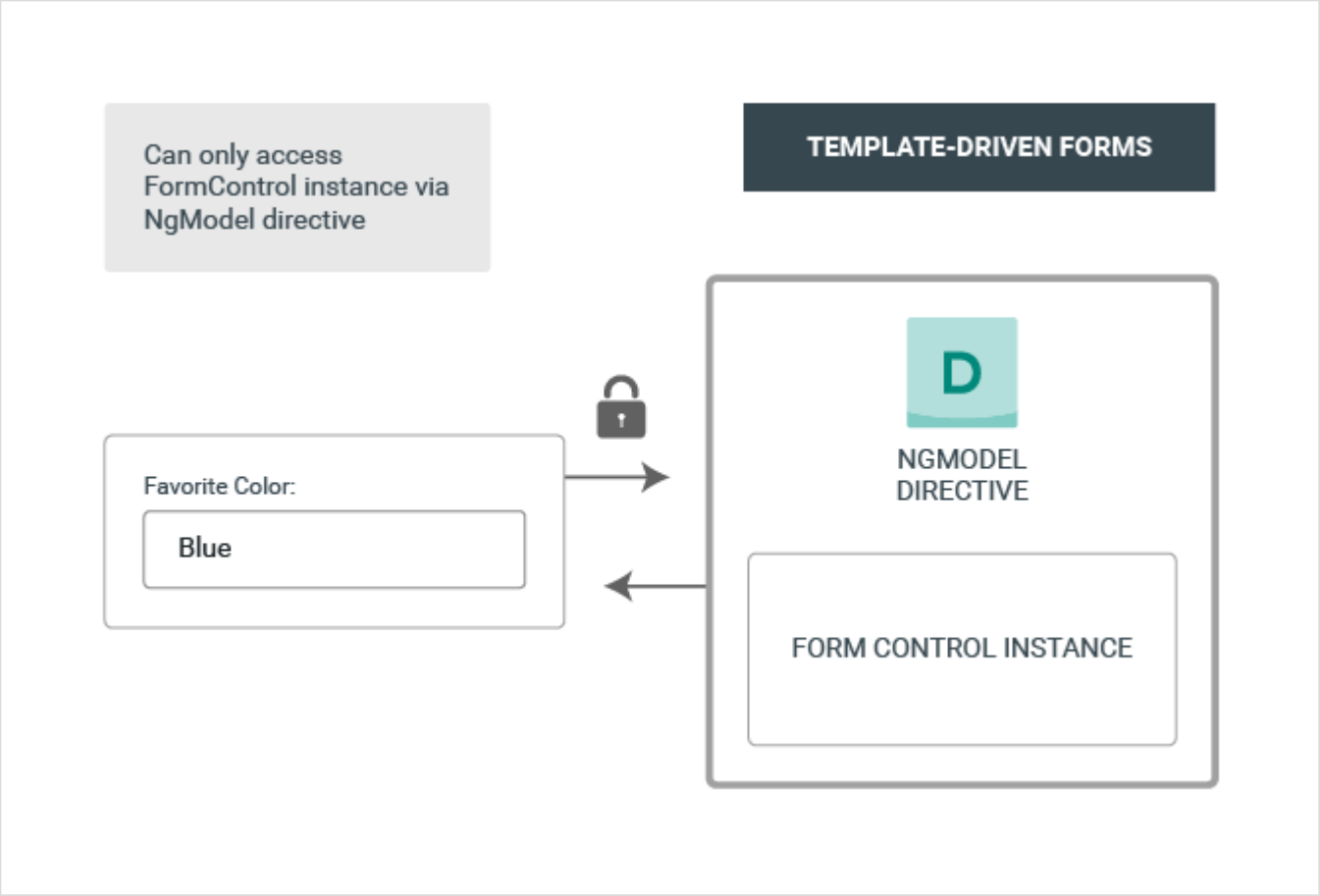
Setup in template-driven forms

Here's the same component with an input field for a single control implemented using template-driven forms.

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

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

In template-driven forms, the source of truth is the template.



The abstraction of the form model promotes simplicity over structure. The template-driven form directive `NgModel` is responsible for creating and managing the `FormControl` instance for a given form element. It's less explicit, but you no longer have direct control over the form model.

Data flow in forms

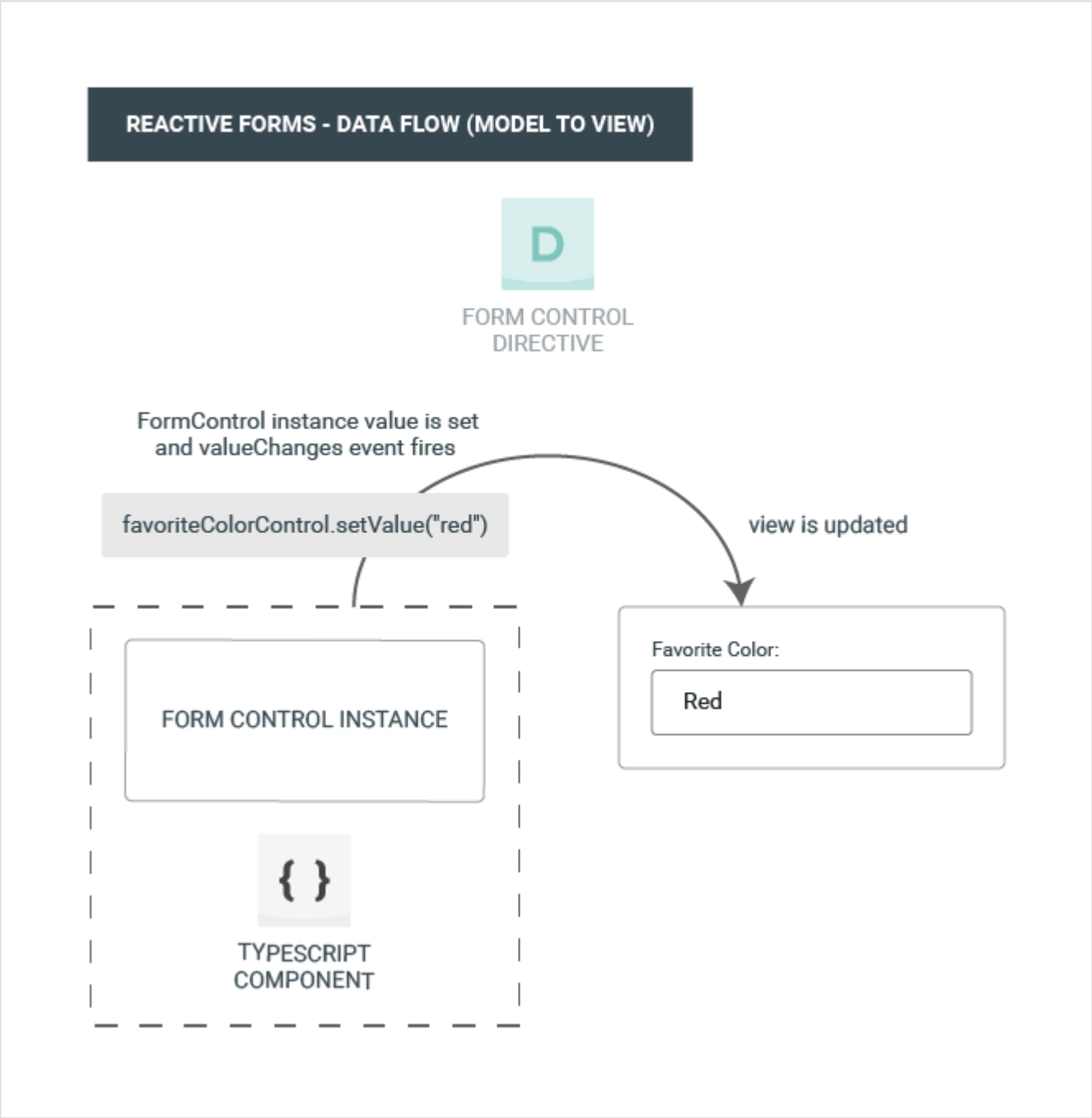
When building forms in Angular, it's important to understand how the framework handles data flowing from the user or from programmatic changes. Reactive and template-driven forms follow two different strategies when handling form input. The data flow examples below begin with the favorite color input field example from above, and then show how changes to favorite color are handled in reactive forms compared to template-driven forms.

Data flow in reactive forms

As described above, in reactive forms each form element in the view is directly linked to a form model (`FormControl` instance). Updates from the view to the model and from the model to the view are synchronous and aren't dependent on the UI rendered. The diagrams below use the same favorite color example to demonstrate how data flows when an input field's value is changed from the view and then from the model.

The steps below outline the data flow from view to model.

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.

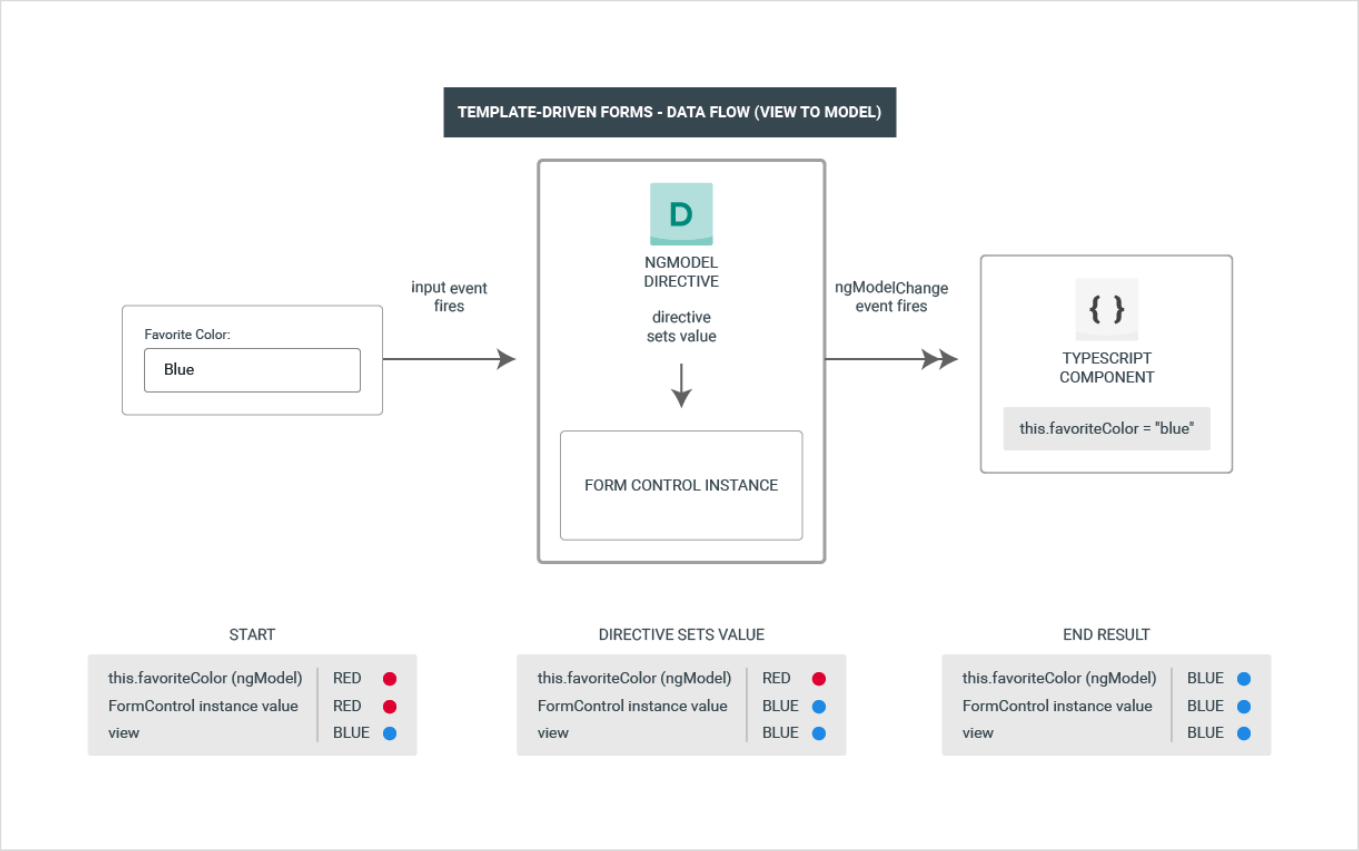


The steps below outline the data flow from model to view.

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.

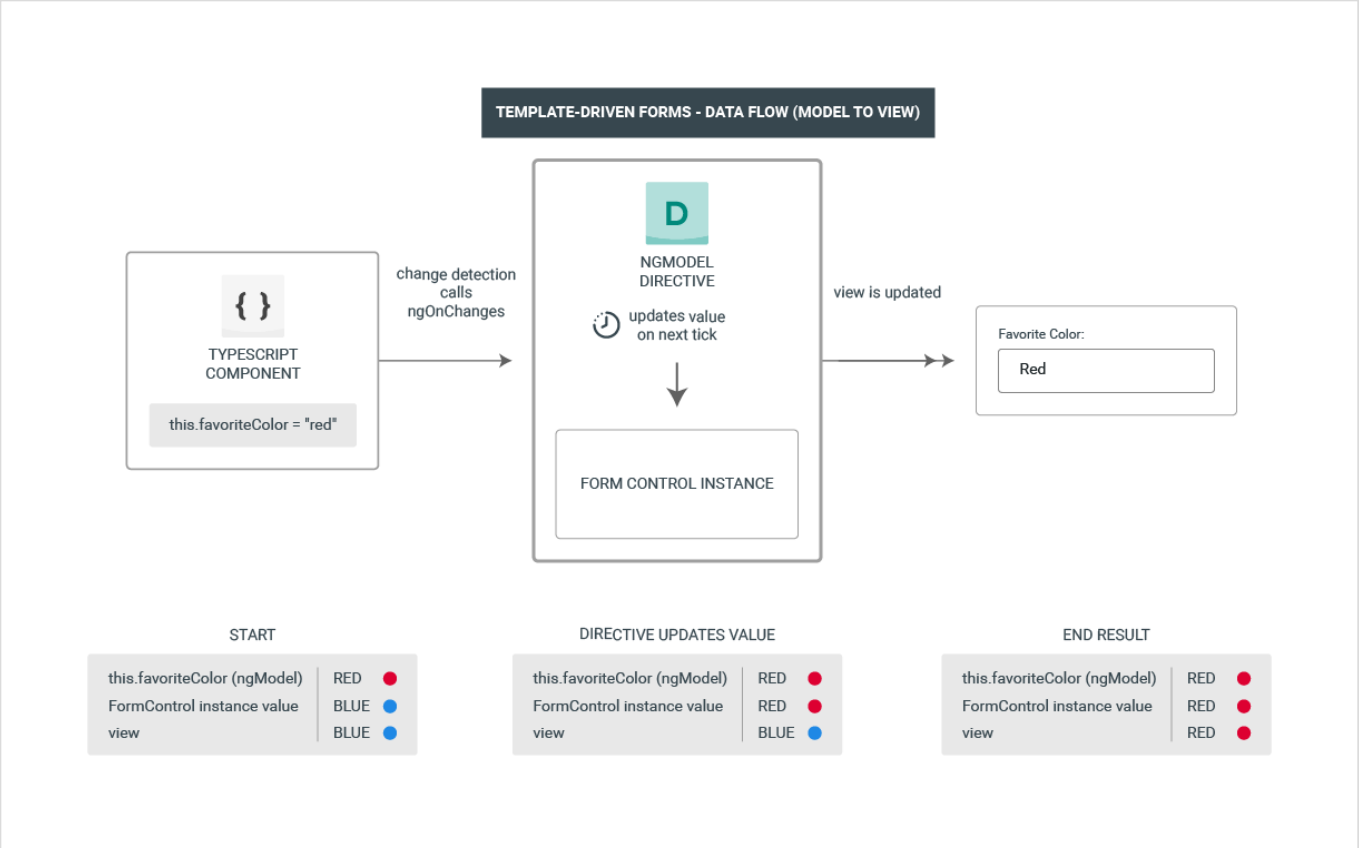
Data flow in template-driven forms

In template-driven forms, each form element is linked to a directive that manages the form model internally. The diagrams below use the same favorite color example to demonstrate how data flows when an input field's value is changed from the view and then from the model.



The steps below outline the data flow from view to model when the input value changes from *Red* to *Blue*.

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 steps below outline the data flow from model to view when the `favoriteColor` changes from *Blue* to *Red*.

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.

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.

- **Reactive forms** define custom validators as **functions** that receive a control to validate.
- **Template-driven forms** are 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 and 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 easy 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 mentioned earlier to verify the data flows from view to model and model to view for a reactive form.

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

Favorite color test - view to model

```
it('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');
});
```

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. Assert that the component's `favoriteColorControl` value matches the value from the input.

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

Favorite color test - model to view

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

  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 `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.

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

```
it('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

```
it('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.

Mutability

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

- **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. This provides one way for change detection to be more efficient because it only needs to update on unique changes. It also follows reactive patterns that 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 examples above using 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.

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** provide access to low-level APIs and synchronous access to the form model, making creating large-scale forms easier.
- **Template-driven** forms focus on simple scenarios, are not as reusable, abstract away the low-level APIs, and provide asynchronous access to the form model. The abstraction with template-driven forms also surfaces in testing, where testing reactive forms requires less setup and no dependence on the change detection cycle when updating and validating the form and data models during testing.

Final thoughts

Choosing a strategy begins with understanding the strengths and weaknesses of the options presented. Low-level API and form model access, predictability, mutability, straightforward validation and testing strategies, and scalability are all important considerations in choosing the infrastructure you use to build your forms in Angular. Template-driven forms are similar to patterns in AngularJS, but they have limitations given the criteria of many modern, large-scale Angular apps. Reactive forms minimize these limitations. Reactive forms integrate with reactive patterns already present in other areas of the Angular architecture, and complement those requirements well.

Next steps

To learn more about reactive forms, see the following guides:

- Reactive Forms
- Form Validation
- Dynamic Forms

To learn more about template-driven forms, see the following guides:

- Template-driven Forms
- Form Validation