**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.

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

**<input type="text" [formControl]="favoriteColorControl">**

export class FavoriteColorComponent {

favoriteColorControl = new FormControl('Red');

}

In reactive forms each form element in the view is directly linked to a form model using (FormControl Directive ). Updates from the view to the model and from the model to the view are synchronous and aren't dependent on the UI rendered.

**Data Flow in Reactive Form**

**From the view to the model**

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

1. When the user types a value into the input element, in above case the favorite color *Blue*.
2. The form input element emits an "input" event with the latest value.
3. The control value accessor (ControlValueAccessor instance) listening for events on the form input element immediately relays the new value to the [FormControl](https://angular.io/api/forms/FormControl) instance.
4. The [FormControl](https://angular.io/api/forms/FormControl) instance emits the new value through the valueChanges observable.
5. Any subscribers to the valueChanges observable receive the new value.

**From the Model to the view**

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

1. When user set the value of model using formControl.setValue() which update the FormControl directive value.
2. The [FormControl](https://angular.io/api/forms/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.

**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.

In The template-driven form directive **NgModel directive** 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.

<input type="text" [([ngModel](https://angular.io/api/forms/NgModel))]="favoriteColor">

export class FavoriteColorComponent {

favoriteColor = '';

}

**Data Flow in Reactive Form**

**From the view to the model**

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

1. When 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](https://angular.io/api/forms/FormControl) instance.
4. The [FormControl](https://angular.io/api/forms/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()](https://angular.io/api/forms/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.

**From the Model to the view**

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

1. The favoriteColor value is updated in the component. favoriteColor = 'RED';
2. Change detection begins.
3. During change detection, the **ngOnChanges** lifecycle hook is called on the [NgModel](https://angular.io/api/forms/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](https://angular.io/api/forms/FormControl) instance.
5. Change detection completes.
6. On the next tick, the task to set the [FormControl](https://angular.io/api/forms/FormControl) instance value is executed.
7. The [FormControl](https://angular.io/api/forms/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 on the form input element updates the element with the new value.