Angular

Qinshift Academy

Reactive forms

Qinshift Academy

What are reactive forms?

Reactive forms provide a model-driven approach to handling form inputs whose values change over time. This guide shows you how to create and update a basic form control, progress to using multiple controls in a group, validate form values, and create dynamic forms where you can add or remove controls at run time.

Overview of reactive forms

Reactive forms use an explicit and immutable approach to managing the state of a form at a given point in time. Each change to the form state returns a new state, which maintains the integrity of the model between changes. Reactive forms are built around observable streams, where form inputs and values are provided as streams of input values, which can be accessed synchronously.

Reactive forms also provide a straightforward path to testing because you are assured that your data is consistent and predictable when requested. Any consumers of the streams have access to manipulate that data safely.

Reactive forms differ from template-driven forms in distinct ways. Reactive forms provide synchronous access to the data model, immutability with observable operators, and change tracking through observable streams.

Template-driven forms let direct access modify data in your template, but are less explicit than reactive forms because they rely on directives embedded in the template, along with mutable data to track changes asynchronously. See the Forms Overview for detailed comparisons between the two paradigms.

Adding a basic form control

There are three steps to using form controls.

Register the reactive forms module in your application. This module declares the reactive-form directives that you need to use reactive forms.

1. Generate a new component and instantiate a new FormControl.
2. Register the FormControl in the template.
3. You can then display the form by adding the component to the template.

The following examples show how to add a single form control. In the example, the user enters their name into an input field, captures that input value, and displays the current value of the form control element.

Import the ReactiveFormsModule

To use reactive form controls, import ReactiveFormsModule from the @angular/forms package and add it to your NgModule's imports array.

```
import {ReactiveFormsModule} from '@angular/forms';
...
@NgModule({
...
  imports: [
...
    // other imports ...
    ReactiveFormsModule,
  ],
...
})
export class AppModule {}
```

Generate a new component with a FormControl

Use the CLI command ng generate component to generate a component in your project to host the control.

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

@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css'],
})
export class NameEditorComponent {
  name = new FormControl('');
...
}
```

Use the constructor of FormControl to set its initial value, which in this case is an empty string. By creating these controls in your component class, you get immediate access to listen for, update, and validate the state of the form input.

Register the control in the template

After you create the control in the component class, you must associate it with a form control element in the template. Update the template with the form control using the formControl binding provided by FormControlDirective, which is also included in the ReactiveFormsModule.

```
<label for="name">Name: </label>
<input id="name" type="text" [formControl]="name">
```

Using the template binding syntax, the form control is now registered to the name input element in the template. The form control and DOM element communicate with each other: the view reflects changes in the model, and the model reflects changes in the view.

Grouping form controls

Forms typically contain several related controls. Reactive forms provide two ways of grouping multiple related controls into a single input form.

Form group

Defines a form with a fixed set of controls that you can manage together. Form group basics are discussed in this section. You can also nest form groups to create more complex forms.

Form array

Defines a dynamic form, where you can add and remove controls at run time. You can also nest form arrays to create more complex forms. For more about this option, see Creating dynamic forms.

Just as a form control instance gives you control over a single input field, a form group instance tracks the form state of a group of form control instances (for example, a form). Each control in a form group instance is tracked by name when creating the form group. The following example shows how to manage multiple form control instances in a single group.

Form Group example

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css'],
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
...
  });
...
}
```

The individual form controls are now collected within a group. A FormGroup instance provides its model value as an object reduced from the values of each control in the group. A form group instance has the same properties (such as value and untouched) and methods (such as setValue()) as a form control instance.

Form Group example

A form group tracks the status and changes for each of its controls, so if one of the controls changes, the parent control also emits a new status or value change. The model for the group is maintained from its members. After you define the model, you must update the template to reflect the model in the view.

```
<form [formGroup]="profileForm">
  <label for="first-name">First Name: </label>
  <input id="first-name" type="text" formControlName="firstName">

  <label for="last-name">Last Name: </label>
  <input id="last-name" type="text" formControlName="lastName">
...
</form>
```

Just as a form group contains a group of controls, the profileForm FormGroup is bound to the form element with the FormGroup directive, creating a communication layer between the model and the form containing the inputs. The formControlName input provided by the FormControlName directive binds each individual input to the form control defined in FormGroup. The form controls communicate with their respective elements. They also communicate changes to the form group instance, which provides the source of truth for the model value.

Saving form data

The ProfileEditor component accepts input from the user, but in a real scenario you want to capture the form value and make it available for further processing outside the component. The FormGroup directive listens for the submit event emitted by the form element and emits an ngSubmit event that you can bind to a callback function. Add an ngSubmit event listener to the form tag with the onSubmit() callback method.

<form [formGroup]="profileForm" (ngSubmit)=“onSubmit()">

The onSubmit() method in the ProfileEditor component captures the current value of profileForm.
Use EventEmitter to keep the form encapsulated and to provide the form value outside the component.
The following example uses console.warn to log a message to the browser console.

```
onSubmit() {
  // TODO: Use EventEmitter with form value
  console.warn(this.profileForm.value);
}
```

Questions?

Trainer Name

Trainer

trainer@mail.com

Assistant Name

Assistant

assistant@mail.com

Qinshift Academy