Reactive forms

Contents ›

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.

> Try this Reactive Forms live-example / download example.

Prerequisites

Before going further into reactive forms, you should have a basic understanding of the following:

- TypeScript programming.
- Angular app-design fundamentals, as described in Angular Concepts.
- The form-design concepts that are presented in Introduction to Forms.

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 more predictability with synchronous access to the data model, immutability with observable operators, and change tracking through observable streams.

Template-driven forms allow direct access to 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.

1. Register the reactive forms module in your app. This module declares the reactive-form directives that you need to use reactive forms.
2. Generate a new `FormControl` instance and save it in the component.
3. Register the `FormControl` in the template.

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.

Register the reactive forms module

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

src/app/app.module.ts (excerpt)

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Generate a new `FormControl`

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

```
ng generate component NameEditor
```

To register a single form control, import the `FormControl` class and create a new instance of `FormControl` to save as a class property.

src/app/name-editor/name-editor.component.ts

```
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`.

src/app/name-editor/name-editor.component.html

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

- For a summary of the classes and directives provided by `ReactiveFormsModule`, see the Reactive forms API section below.
- For complete syntax details of these classes and directives, see the API reference documentation for the Forms package.

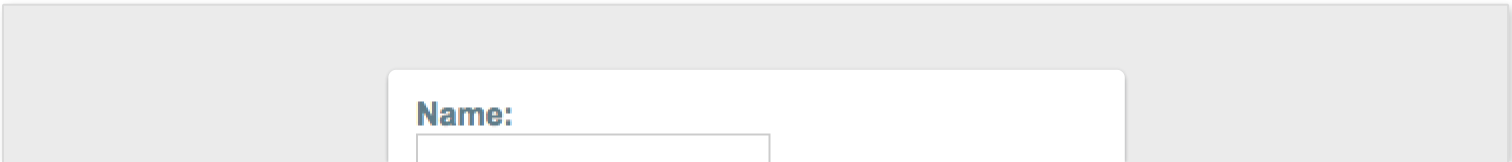
Using the template binding syntax, the form control is now registered to the `name` input element in the template. The form control and DOM element communicate with each other: the view reflects changes in the model, and the model reflects changes in the view.

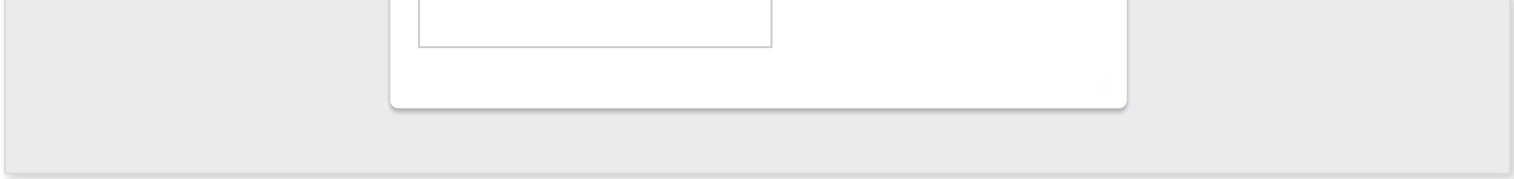
Display the component

The form control assigned to `name` is displayed when the component is added to a template.

src/app/app.component.html (name editor)

```
<app-name-editor></app-name-editor>
```





Displaying a form control value

You can display the value in the following ways.

- Through the `valueChanges` observable where you can listen for changes in the form's value in the template using `AsyncPipe` or in the component class using the `subscribe()` method.
- With the `value` property, which gives you a snapshot of the current value.

The following example shows you how to display the current value using interpolation in the template.

src/app/name-editor/name-editor.component.html (control value)

```html
<p>
  Value: {{ name.value }}
</p>
```

The displayed value changes as you update the form control element.

Reactive forms provide access to information about a given control through properties and methods provided with each instance. These properties and methods of the underlying AbstractControl class are used to control form state and determine when to display messages when handling input validation.

Read about other `FormControl` properties and methods in the API Reference.

Replacing a form control value

Reactive forms have methods to change a control's value programmatically, which gives you the flexibility to update the value without user interaction. A form control instance provides a `setValue()` method that updates the value of the form control and validates the structure of the value provided against the control's structure. For example, when retrieving form data from a backend API or service, use the `setValue()` method to update the control to its new value, replacing the old value entirely.

The following example adds a method to the component class to update the value of the control to *Nancy* using the `setValue()` method.

src/app/name-editor/name-editor.component.ts (update value)

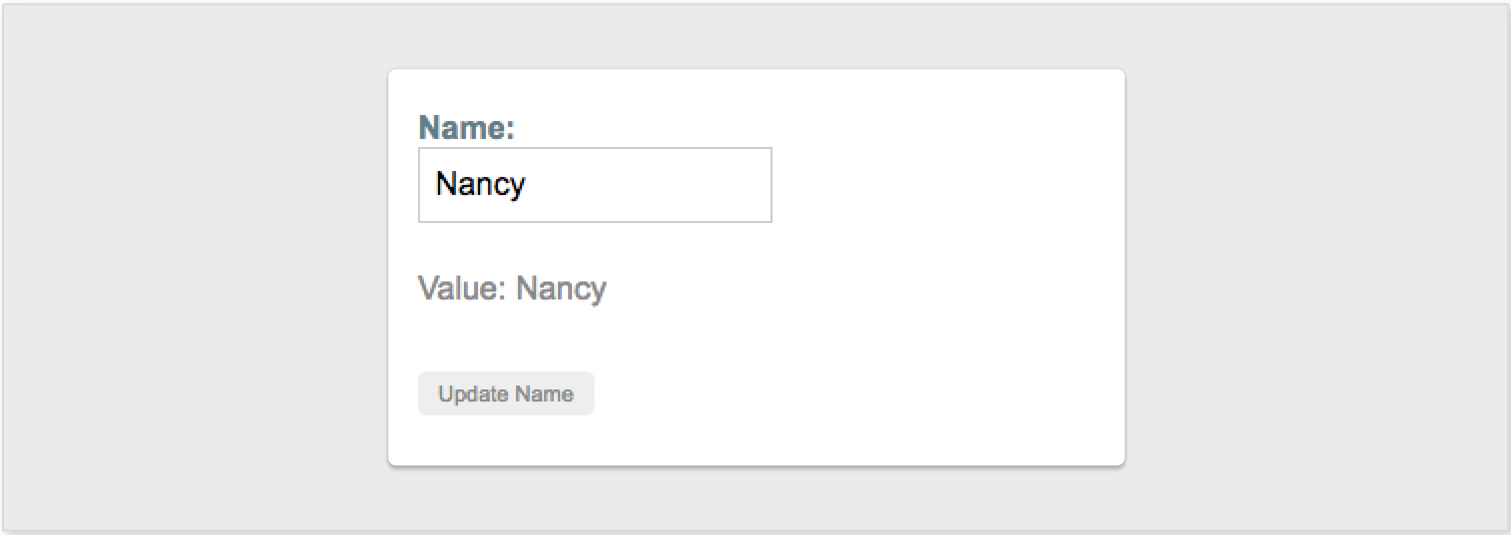
```typescript
updateName() {
  this.name.setValue('Nancy');
}
```

Update the template with a button to simulate a name update. When you click the **Update Name** button, the value entered in the form control element is reflected as its current value.

src/app/name-editor/name-editor.component.html (update value)

```
<p>
  <button (click)="updateName()">Update Name</button>
</p>
```

The form model is the source of truth for the control, so when you click the button, the value of the input is changed within the component class, overriding its current value.



Note: In this example, you're using a single control. When using the `setValue()` method with a form group or form array instance, the value needs to match the structure of the group or array.

Grouping form controls

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

- A 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.
- A 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 below.

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.

Generate a `ProfileEditor` component and import the `FormGroup` and `FormControl` classes from the `@angular/forms` package.

```
ng generate component ProfileEditor
```

src/app/profile-editor/profile-editor.component.ts (imports)

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

To add a form group to this component, take the following steps.

1. Create a `FormGroup` instance.
2. Associate the `FormGroup` model and view.
3. Save the form data.

Create a FormGroup instance

Create a property in the component class named `profileForm` and set the property to a new form group instance. To initialize the form group, provide the constructor with an object of named keys mapped to their control.

For the profile form, add two form control instances with the names `firstName` and `lastName`.

src/app/profile-editor/profile-editor.component.ts (form group)

```
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.

Associate the FormGroup model and view

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.

src/app/profile-editor/profile-editor.component.html (template form group)

```
<form [formGroup]="profileForm">

  <label>
```

```
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>

</form>
```

Note that just as a form group contains a group of controls, the *profile form* `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.

Save form data

The `ProfileEditor` component accepts input from the user, but in a real scenario you want to capture the form value and make 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.

src/app/profile-editor/profile-editor.component.html (submit event)

```
<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.

src/app/profile-editor/profile-editor.component.ts (submit method)

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

The `submit` event is emitted by the `form` tag using the native DOM event. You trigger the event by clicking a button with `submit` type. This allows the user to press the **Enter** key to submit the completed form.

Use a `button` element to add a button to the bottom of the form to trigger the form submission.

src/app/profile-editor/profile-editor.component.html (submit button)

```html
<button type="submit" [disabled]="!profileForm.valid">Submit</button>
```

Note: The button in the snippet above also has a `disabled` binding attached to it to disable the button when `profileForm` is invalid. You aren't performing any validation yet, so the button is always enabled. Basic form validation is covered in the Validating form input section.

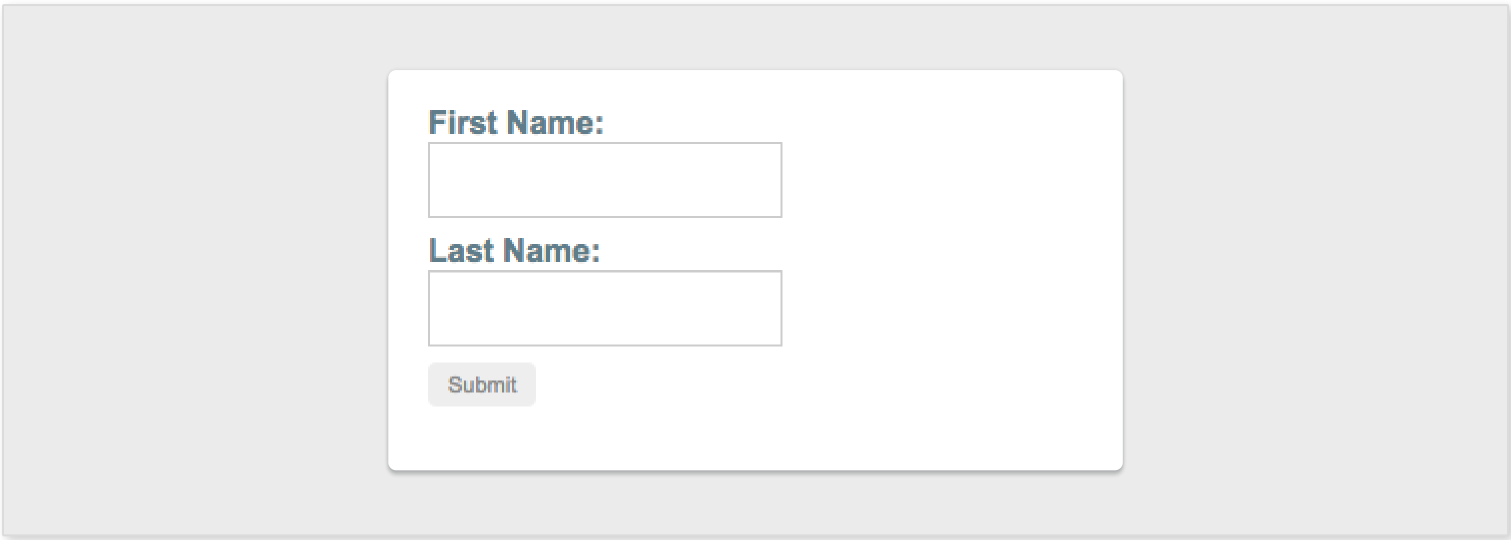
Display the component

To display the `ProfileEditor` component that contains the form, add it to a component template.

src/app/app.component.html (profile editor)

```html
<app-profile-editor></app-profile-editor>
```

`ProfileEditor` allows you to manage the form control instances for the `firstName` and `lastName` controls within the form group instance.



Creating nested form groups

Form groups can accept both individual form control instances and other form group instances as children. This makes composing complex form models easier to maintain and logically group together.

When building complex forms, managing the different areas of information is easier in smaller sections. Using a nested form group instance allows you to break large forms groups into smaller, more manageable ones.

To make more complex forms, use the following steps.

1. Create a nested group.
2. Group the nested form in the template.

Some types of information naturally fall into the same group. A name and address are typical examples of such nested groups, and are used in the following examples.

Create a nested group

To create a nested group in `profileForm`, add a nested `address` element to the form group instance.

src/app/profile-editor/profile-editor.component.ts (nested form group)

```
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(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });
}
```

In this example, `address group` combines the current `firstName` and `lastName` controls with the new `street`, `city`, `state`, and `zip` controls. Even though the `address` element in the form group is a child of the overall `profileForm` element in the form group, the same rules apply with value and status changes. Changes in status and value from the nested form group propagate to the parent form group, maintaining consistency with the overall model.

Group the nested form in the template

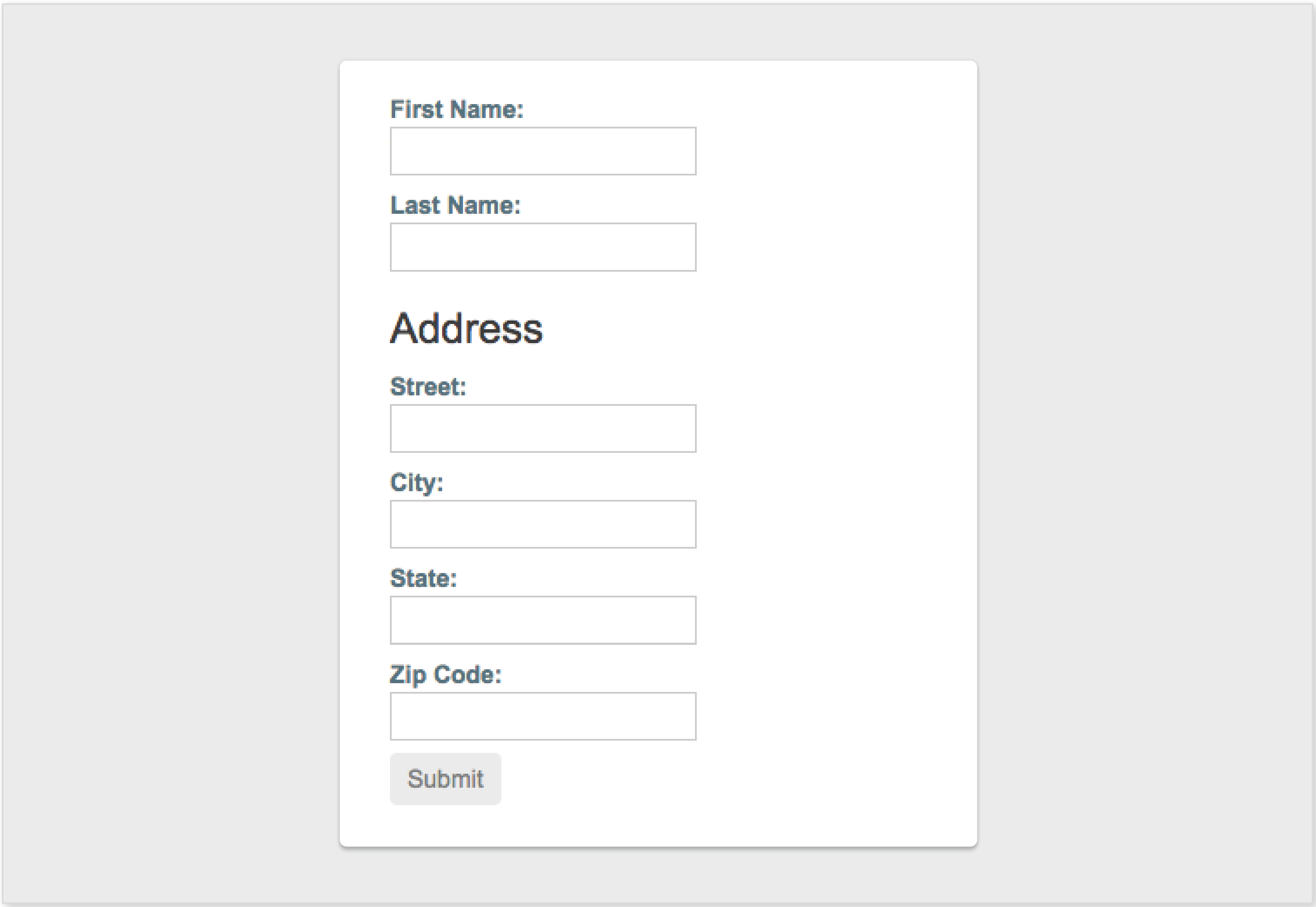
After you update the model in the component class, update the template to connect the form group instance and its input elements.

Add the `address` form group containing the `street`, `city`, `state`, and `zip` fields to the `ProfileEditor` template.

src/app/profile-editor/profile-editor.component.html (template nested form group)

```
<div formGroupName="address">
  <h3>Address</h3>

  <label>
    Street:
    <input type="text" formControlName="street">
  </label>

  <label>
    City:
    <input type="text" formControlName="city">
  </label>

  <label>
    State:
    <input type="text" formControlName="state">
  </label>

  <label>
    Zip Code:
    <input type="text" formControlName="zip">
  </label>
</div>
```

The `ProfileEditor` form is displayed as one group, but the model is broken down further to represent the logical grouping areas.



Tip Display the value for the form group instance in the component template using the `value` property and `JsonPipe`.

Updating parts of the data model

When updating the value for a form group instance that contains multiple controls, you may only want to update parts of the model. This section covers how to update specific parts of a form control data model.

There are two ways to update the model value:

- Use the `setValue()` method to set a new value for an individual control. The `setValue()` method strictly adheres to the structure of the form group and replaces the entire value for the control.
- Use the `patchValue()` method to replace any properties defined in the object that have changed in the form model.

The strict checks of the `setValue()` method help catch nesting errors in complex forms, while `patchValue()` fails silently on those errors.

In `ProfileEditorComponent`, use the `updateProfile` method with the example below to update the first name and street address for the user.

src/app/profile-editor/profile-editor.component.ts (patch value)

```
updateProfile() {
  this.profileForm.patchValue({
    firstName: 'Nancy',
    address: {
      street: '123 Drew Street'
    }
  });
}
```

Simulate an update by adding a button to the template to update the user profile on demand.

src/app/profile-editor/profile-editor.component.html (update value)

```
<p>
  <button (click)="updateProfile()">Update Profile</button>
</p>
```

When a user clicks the button, the `profileForm` model is updated with new values for `firstName` and `street`. Notice that `street` is provided in an object inside the `address` property. This is necessary because the `patchValue()` method applies the update against the model structure. `PatchValue()` only updates properties that the form model defines.

Using the FormBuilder service to generate controls

Creating form control instances manually can become repetitive when dealing with multiple forms. The `FormBuilder` service provides convenient methods for generating controls.

Use the following steps to take advantage of this service.

1. Import the `FormBuilder` class.
2. Inject the `FormBuilder` service.
3. Generate the form contents.

The following examples show how to refactor the `ProfileEditor` component to use the form builder service to create form control and form group instances.

Import the FormBuilder class

Import the `FormBuilder` class from the `@angular/forms` package.

src/app/profile-editor/profile-editor.component.ts (import)

```
import { FormBuilder } from '@angular/forms';
```

Inject the FormBuilder service

The `FormBuilder` service is an injectable provider that is provided with the reactive forms module. Inject this dependency by adding it to the component constructor.

src/app/profile-editor/profile-editor.component.ts (constructor)

```
constructor(private fb: FormBuilder) { }
```

Generate form controls

The `FormBuilder` service has three methods: `control()`, `group()`, and `array()`. These are factory methods for generating instances in your component classes including form controls, form groups, and form arrays.

Use the `group` method to create the `profileForm` controls.

src/app/profile-editor/profile-editor.component.ts (form builder)

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

  constructor(private fb: FormBuilder) { }
}
```

In the example above, you use the `group()` method with the same object to define the properties in the model. The value for each control name is an array containing the initial value as the first item in the array.

> **Tip** You can define the control with just the initial value, but if your controls need sync or async validation, add sync and async validators as the second and third items in the array.

Compare using the form builder to creating the instances manually.

src/app/profile-editor/profile-editor.component.ts (instances) src/app/profile-editor/profile-edit

```
profileForm = new FormGroup({
  firstName: new FormControl(''),
  lastName: new FormControl(''),
  address: new FormGroup({
    street: new FormControl(''),
    city: new FormControl(''),
    state: new FormControl(''),
    zip: new FormControl('')
  })
});
```

Validating form input

Form validation is used to ensure that user input is complete and correct. This section covers adding a single validator to a form control and displaying the overall form status. Form validation is covered more extensively in the Form Validation guide.

Use the following steps to add form validation.

1. Import a validator function in your form component.
2. Add the validator to the field in the form.
3. Add logic to handle the validation status.

The most common validation is making a field required. The following example shows how to add a required validation to the `firstName` control and display the result of validation.

Import a validator function

Reactive forms include a set of validator functions for common use cases. These functions receive a control to validate against and return an error object or a null value based on the validation check.

Import the `Validators` class from the `@angular/forms` package.

src/app/profile-editor/profile-editor.component.ts (import)

```
import { Validators } from '@angular/forms';
```

Make a field required

In the `ProfileEditor` component, add the `Validators.required` static method as the second item in the array for the `firstName` control.

src/app/profile-editor/profile-editor.component.ts (required validator)

```
profileForm = this.fb.group({
  firstName: ['', Validators.required],
  lastName: [''],
  address: this.fb.group({
    street: [''],
    city: [''],
    state: [''],
    zip: ['']
  }),
});
```

HTML5 has a set of built-in attributes that you can use for native validation, including `required`, `minlength`, and `maxlength`. You can take advantage of these optional attributes on your form input elements. Add the `required` attribute to the `firstName` input element.

src/app/profile-editor/profile-editor.component.html (required attribute)

```
<input type="text" formControlName="firstName" required>
```

Caution: Use these HTML5 validation attributes *in combination with* the built-in validators provided by Angular's reactive forms. Using these in combination prevents errors when the expression is changed after the template has been checked.

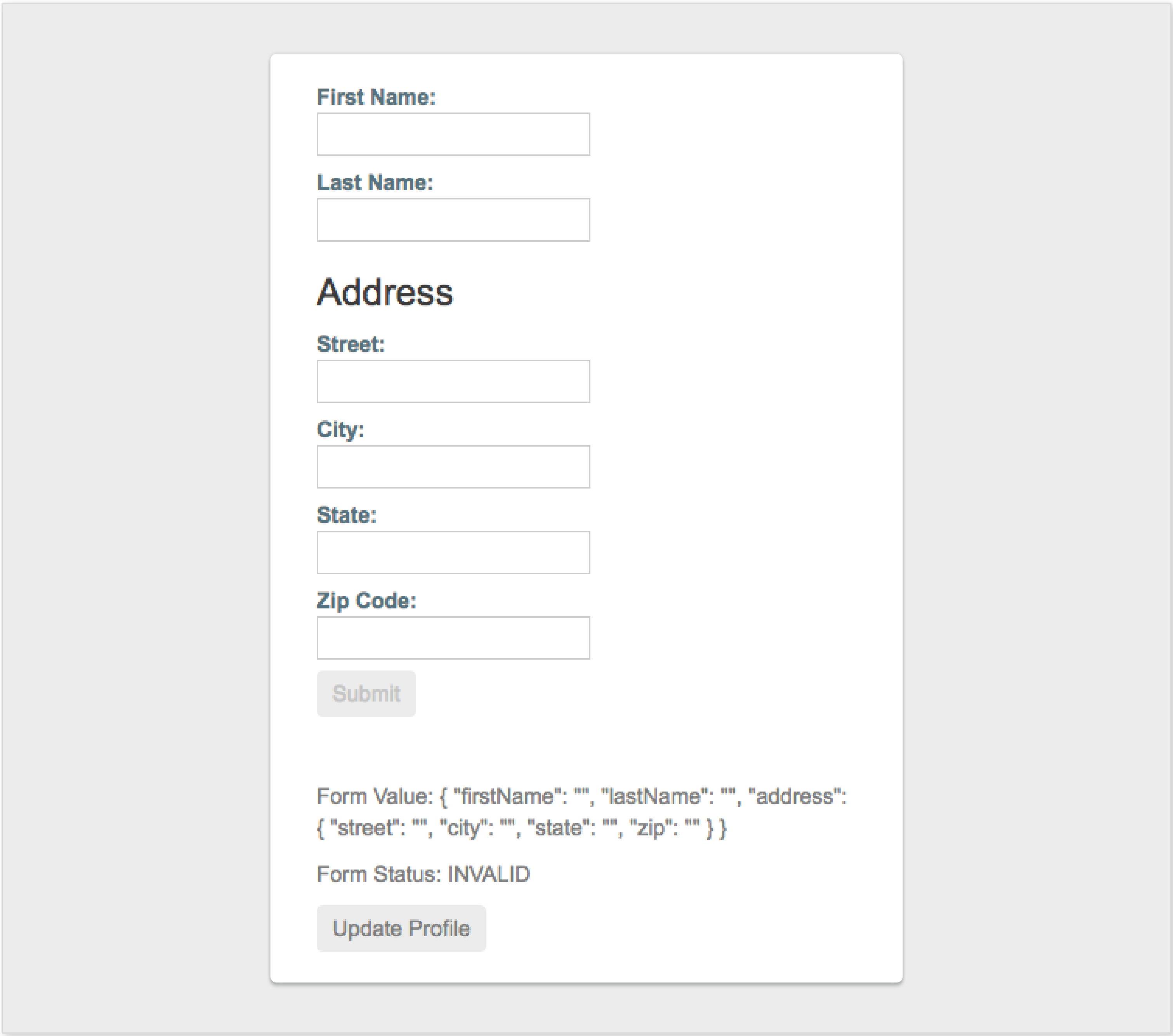
Display form status

When you add a required field to the form control, its initial status is invalid. This invalid status propagates to the parent form group element, making its status invalid. Access the current status of the form group instance through its `status` property.

Display the current status of `profileForm` using interpolation.

src/app/profile-editor/profile-editor.component.html (display status)

```
<p>
  Form Status: {{ profileForm.status }}
</p>
```



Form Value: { "firstName": "", "lastName": "", "address": { "street": "", "city": "", "state": "", "zip": "" } }

Form Status: INVALID

The **Submit** button is disabled because `profileForm` is invalid due to the required `firstName` form control. After you fill out the `firstName` input, the form becomes valid and the **Submit** button is enabled.

For more on form validation, visit the Form Validation guide.

Creating dynamic forms

`FormArray` is an alternative to `FormGroup` for managing any number of unnamed controls. As with form group instances, you can dynamically insert and remove controls from form array instances, and the form array instance value and validation status is calculated from its child controls. However, you don't need to define a key for each control by name, so this is a great option if you don't know the number of child values in advance.

To define a dynamic form, take the following steps.

1. Import the `FormArray` class.
2. Define a `FormArray` control.
3. Access the `FormArray` control with a getter method.
4. Display the form array in a template.

The following example shows you how to manage an array of *aliases* in `ProfileEditor`.

Import the FormArray class

Import the `FormArray` class from `@angular/forms` to use for type information. The `FormBuilder` service is ready to create a `FormArray` instance.

src/app/profile-editor/profile-editor.component.ts (import)

```
import { FormArray } from '@angular/forms';
```

Define a FormArray control

You can initialize a form array with any number of controls, from zero to many, by defining them in an array. Add an `aliases` property to the form group instance for `profileForm` to define the form array.

Use the `FormBuilder.array()` method to define the array, and the `FormBuilder.control()` method to populate the array with an initial control.

src/app/profile-editor/profile-editor.component.ts (aliases form array)

```
profileForm = this.fb.group({
  firstName: ['', Validators.required],
  lastName: [''],
  address: this.fb.group({
    street: [''],
    city: [''],
    state: [''],
    zip: ['']
  }),
  aliases: this.fb.array([
    this.fb.control('')
  ])
});
```

The aliases control in the form group instance is now populated with a single control until more controls are added dynamically.

Access the FormArray control

A getter provides easy access to the aliases in the form array instance compared to repeating the `profileForm.get()` method to get each instance. The form array instance represents an undefined number of controls in an array. It's convenient to access a control through a getter, and this approach is easy to repeat for additional controls.

Use the getter syntax to create an `aliases` class property to retrieve the alias's form array control from the parent form group.

src/app/profile-editor/profile-editor.component.ts (aliases getter)

```
get aliases() {
  return this.profileForm.get('aliases') as FormArray;
}
```

Note: Because the returned control is of the type `AbstractControl`, you need to provide an explicit type to access the method syntax for the form array instance.

Define a method to dynamically insert an alias control into the alias's form array. The `FormArray.push()` method inserts the control as a new item in the array.

src/app/profile-editor/profile-editor.component.ts (add alias)

```
addAlias() {
  this.aliases.push(this.fb.control(''));
}
```

In the template, each control is displayed as a separate input field.

Display the form array in the template

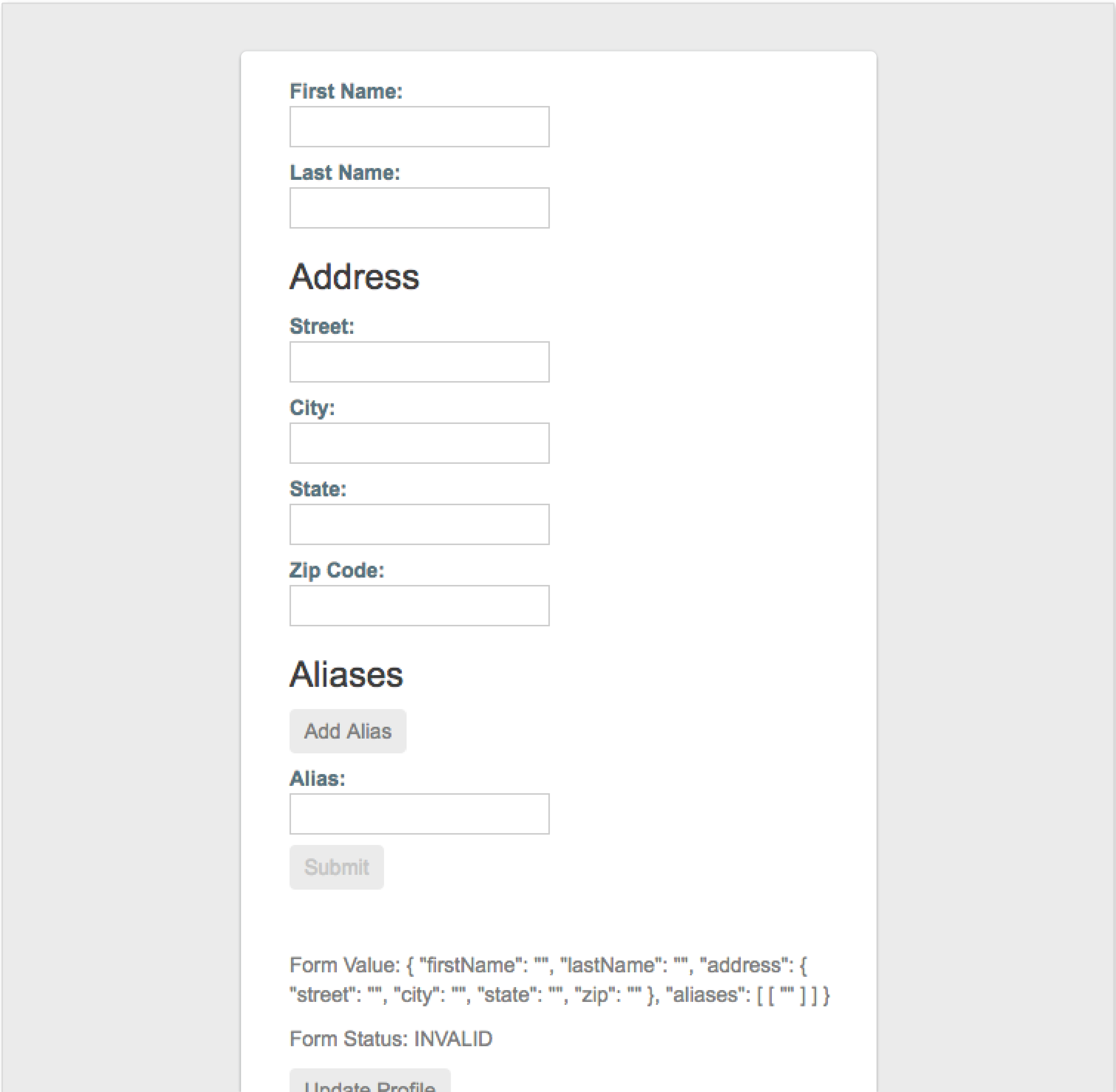
To attach the aliases from your form model, you must add it to the template. Similar to the `formGroupName` input provided by `FormGroupNameDirective`, `formArrayName` binds communication from the form array instance to the template with `FormArrayNameDirective`.

Add the template HTML below after the `<div>` closing the `formGroupName` element.

src/app/profile-editor/profile-editor.component.html (aliases form array template)

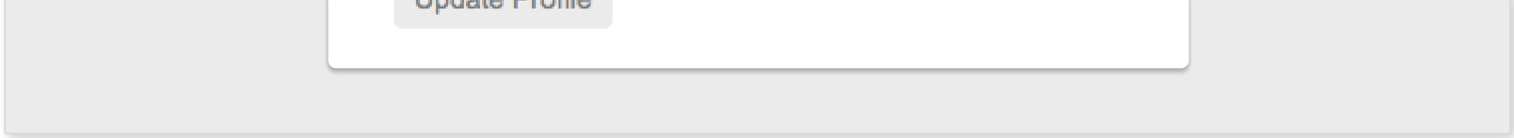
```
<div formArrayName="aliases">
  <h3>Aliases</h3> <button (click)="addAlias()">Add Alias</button>

  <div *ngFor="let alias of aliases.controls; let i=index">
    <!-- The repeated alias template -->
    <label>
      Alias:
      <input type="text" [formControlName]="i">
    </label>
  </div>
</div>
```

The `*ngFor` directive iterates over each form control instance provided by the aliases form array instance. Because form array elements are unnamed, you assign the index to the `i` variable and pass it to each control to bind it to the `formControlName` input.



Form Value: { "firstName": "", "lastName": "", "address": { "street": "", "city": "", "state": "", "zip": "" }, "aliases": [[""]] }

Form Status: INVALID



Each time a new alias instance is added, the new form array instance is provided its control based on the index. This allows you to track each individual control when calculating the status and value of the root control.

Add an alias

Initially, the form contains one `Alias` field. To add another field, click the **Add Alias** button. You can also validate the array of aliases reported by the form model displayed by `Form Value` at the bottom of the template.

> **Note:** Instead of a form control instance for each alias, you can compose another form group instance with additional fields. The process of defining a control for each item is the same.

Reactive forms API summary

The following table lists the base classes and services used to create and manage reactive form controls. For complete syntax details, see the API reference documentation for the Forms package.

Classes

Class	**Description**
`AbstractControl`	The abstract base class for the concrete form control classes `FormControl`, `FormGroup`, and `FormArray`. It provides their common behaviors and properties.
`FormControl`	Manages the value and validity status of an individual form control. It corresponds to an HTML form control such as `<input>` or `<select>`.
`FormGroup`	Manages the value and validity state of a group of `AbstractControl` instances. The group's properties include its child controls. The top-level form in your component is `FormGroup`.
`FormArray`	Manages the value and validity state of a numerically indexed array of `AbstractControl` instances.
`FormBuilder`	An injectable service that provides factory methods for creating control instances.

Directives

Directive	**Description**

`FormControlDirective`	Syncs a standalone `FormControl` instance to a form control element.
`FormControlName`	Syncs `FormControl` in an existing `FormGroup` instance to a form control element by name.
`FormGroupDirective`	Syncs an existing `FormGroup` instance to a DOM element.
`FormGroupName`	Syncs a nested `FormGroup` instance to a DOM element.
`FormArrayName`	Syncs a nested `FormArray` instance to a DOM element.