Reactive Forms

Reactive forms is an Angular technique for creating forms in a *reactive* style. This guide explains reactive forms as you follow the steps to build a "Hero Detail Editor" form.

{@a toc}

Try the Reactive Forms live-example.

You can also run the Reactive Forms Demo version and choose one of the intermediate steps from the "demo picker" at the top.

{@a intro}

Introduction to Reactive Forms

Angular offers two form-building technologies: *reactive* forms and *template-driven* forms. The two technologies belong to the `@angular/forms` library and share a common set of form control classes.

But they diverge markedly in philosophy, programming style, and technique. They even have their own modules: the `ReactiveFormsModule` and the `FormsModule`.

Reactive forms

Angular *reactive* forms facilitate a *reactive style* of programming that favors explicit management of the data flowing between a non-UI *data model* (typically retrieved from a server) and a UI-oriented *form model* that retains the states and values of the HTML controls on screen. Reactive forms offer the ease of using reactive patterns, testing, and validation.

With *reactive* forms, you create a tree of Angular form control objects in the component class and bind them to native form control elements in the component template, using techniques described in this guide.

You create and manipulate form control objects directly in the component class. As the component class has immediate access to both the data model and the form control structure, you can push data model values into the form controls and pull user-changed values back out. The component can observe changes in form control state and react to those changes.

One advantage of working with form control objects directly is that value and validity updates are always synchronous and under your control. You won't encounter the timing issues that sometimes plague a template-

driven form and reactive forms can be easier to unit test.

In keeping with the reactive paradigm, the component preserves the immutability of the *data model*, treating it as a pure source of original values. Rather than update the data model directly, the component extracts user changes and forwards them to an external component or service, which does something with them (such as saving them) and returns a new *data model* to the component that reflects the updated model state.

Using reactive form directives does not require you to follow all reactive priniciples, but it does facilitate the reactive programming approach should you choose to use it.

Template-driven forms

Template-driven forms, introduced in the Template guide, take a completely different approach.

You place HTML form controls (such as `<input>` and `<select>`) in the component template and bind them to *data model* properties in the component, using directives like `ngModel`.

You don't create Angular form control objects. Angular directives create them for you, using the information in your data bindings. You don't push and pull data values. Angular handles that for you with `ngModel`. Angular updates the mutable *data model* with user changes as they happen.

For this reason, the `ngModel` directive is not part of the ReactiveFormsModule.

While this means less code in the component class, template-driven forms are asynchronous which may complicate development in more advanced scenarios.

{@a async-vs-sync}

Async vs. sync

Reactive forms are synchronous. Template-driven forms are asynchronous. It's a difference that matters.

In reactive forms, you create the entire form control tree in code. You can immediately update a value or drill down through the descendents of the parent form because all controls are always available.

Template-driven forms delegate creation of their form controls to directives. To avoid "*changed after checked*" errors, these directives take more than one cycle to build the entire control tree. That means you must wait a tick before manipulating any of the controls from within the component class.

For example, if you inject the form control with a `@ViewChild(NgForm)` query and examine it in the `ngAfterViewInit` lifecycle hook, you'll discover that it has no children. You must wait a tick, using `setTimeout`, before you can extract a value from a control, test its validity, or set it to a new value.

The asynchrony of template-driven forms also complicates unit testing. You must wrap your test block in `async()` or `fakeAsync()` to avoid looking for values in the form that aren't there yet. With reactive forms, everything is available when you expect it to be.

Which is better, reactive or template-driven?

Neither is "better". They're two different architectural paradigms, with their own strengths and weaknesses. Choose the approach that works best for you. You may decide to use both in the same application.

The balance of this *reactive forms* guide explores the *reactive* paradigm and concentrates exclusively on reactive forms techniques. For information on *template-driven forms*, see the *Forms* guide.

In the next section, you'll set up your project for the reactive form demo. Then you'll learn about the Angular form classes and how to use them in a reactive form.

{@a setup}

Setup

Create a new project named `angular-reactive-forms`:

ng new angular-reactive-forms

{@a data-model}

Create a data model

The focus of this guide is a reactive forms component that edits a hero. You'll need a `hero` class and some hero data.

Using the CLI, generate a new class named `data-model`:

ng generate class data-model

And copy the content below:

The file exports two classes and two constants. The `Address` and `Hero` classes define the application *data model*. The `heroes` and `states` constants supply the test data.

{@a create-component}

Create a *reactive forms* component

Generate a new component named `HeroDetail`:

ng generate component HeroDetail

And import:

Next, update the `HeroDetailComponent` class with a `FormControl`. `FormControl` is a directive that allows you to create and manage a `FormControl` instance directly.

Here you are creating a `FormControl` called `name`. It will be bound in the template to an HTML `input` box for the hero name.

A `FormControl` constructor accepts three, optional arguments: the initial data value, an array of validators, and an array of async validators.

This simple control doesn't have data or validators. In real apps, most form controls have both.

This guide touches only briefly on `Validators`. For an in-depth look at them, read the [Form Validation] (guide/form-validation) guide.

{@a create-template}

Create the template

Now update the component's template, with the following markup.

To let Angular know that this is the input that you want to associate to the `name` `FormControl` in the class, you need `[formControl]="name"` in the template on the `<input>`.

Disregard the `form-control` _CSS_ class. It belongs to the Bootstrap CSS library, not Angular. It _styles_ the form but in no way impacts the logic of the form.

{@a import}

Import the *ReactiveFormsModule*

The HeroDetailComponent template uses `formControlName` directive from the `ReactiveFormsModule`.

Do the following two things in `app.module.ts` :

1. Use a JavaScript `import` statement to access the `ReactiveFormsModule` .
2. Add `ReactiveFormsModule` to the `AppModule` 's `imports` list.

{@a update}

Display the *HeroDetailComponent*

Revise the `AppComponent` template so it displays the `HeroDetailComponent` .

{@a essentials}

Essential form classes

It may be helpful to read a brief description of the core form classes.

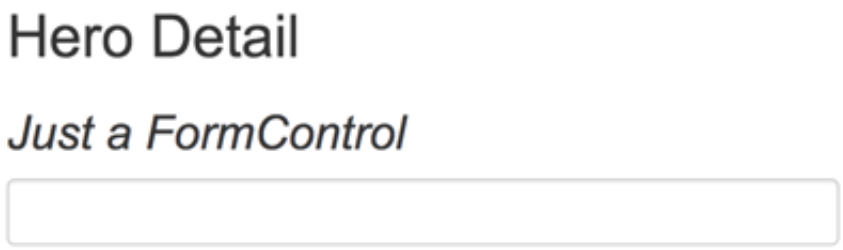
- *AbstractControl* is the abstract base class for the three concrete form control classes: `FormControl` , `FormGroup` , and `FormArray` . It provides their common behaviors and properties, some of which are *observable*.
- *FormControl* tracks the value and validity status of an *individual* form control. It corresponds to an HTML form control such as an input box or selector.
- *FormGroup* tracks 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 a `FormGroup` .
- *FormArray* tracks the value and validity state of a numerically indexed *array* of `AbstractControl` instances.

You'll learn more about these classes as you work through this guide.

Style the app

You used bootstrap CSS classes in the template HTML of both the `AppComponent` and the `HeroDetailComponent` . Add the `bootstrap` *CSS stylesheet* to the head of `styles.css` :

Now that everything is wired up, the browser should display something like this:



{@a formgroup}

Add a FormGroup

Usually, if you have multiple *FormControls*, you'll want to register them within a parent `FormGroup`. This is simple to do. To add a `FormGroup`, add it to the imports section of `hero-detail.component.ts`:

In the class, wrap the `FormControl` in a `FormGroup` called `heroForm` as follows:

Now that you've made changes in the class, they need to be reflected in the template. Update `hero-detail.component.html` by replacing it with the following.

Notice that now the single input is in a `form` element. The `novalidate` attribute in the `<form>` element prevents the browser from attempting native HTML validations.

`formGroup` is a reactive form directive that takes an existing `FormGroup` instance and associates it with an HTML element. In this case, it associates the `FormGroup` you saved as `heroForm` with the form element.

Because the class now has a `FormGroup`, you must update the template syntax for associating the input with the corresponding `FormControl` in the component class. Without a parent `FormGroup`, `[formControl]="name"` worked earlier because that directive can stand alone, that is, it works without being in a `FormGroup`. With a parent `FormGroup`, the `name` input needs the syntax `formControlName=name` in order to be associated with the correct `FormControl` in the class. This syntax tells Angular to look for the parent `FormGroup`, in this case `heroForm`, and then *inside* that group to look for a `FormControl` called `name`.

Disregard the `form-group` _CSS_ class. It belongs to the Bootstrap CSS library, not Angular. Like the `form-control` class, it _styles_ the form but in no way impacts its logic.

The form looks great. But does it work? When the user enters a name, where does the value go?

{@a json}

Taking a look at the form model

The value goes into the ***form model*** that backs the group's `FormControls`. To see the form model, add the following line after the closing `form` tag in the `hero-detail.component.html`:

The `heroForm.value` returns the *form model*. Piping it through the `JsonPipe` renders the model as JSON in the browser:

The initial `name` property value is the empty string. Type into the *name* input box and watch the keystokes appear in the JSON.

Great! You have the basics of a form.

In real life apps, forms get big fast. `FormBuilder` makes form development and maintenance easier.

{@a formbuilder}

Introduction to *FormBuilder*

The `FormBuilder` class helps reduce repetition and clutter by handling details of control creation for you.

To use `FormBuilder`, you need to import it into `hero-detail.component.ts`:

Use it now to refactor the `HeroDetailComponent` into something that's a little easier to read and write, by following this plan:

- Explicitly declare the type of the `heroForm` property to be `FormGroup`; you'll initialize it later.
- Inject a `FormBuilder` into the constructor.
- Add a new method that uses the `FormBuilder` to define the `heroForm`; call it `createForm`.
- Call `createForm` in the constructor.

The revised `HeroDetailComponent` looks like this:

`FormBuilder.group` is a factory method that creates a `FormGroup`. `FormBuilder.group` takes an object whose keys and values are `FormControl` names and their definitions. In this example, the `name` control is defined by its initial data value, an empty string.

Defining a group of controls in a single object makes for a compact, readable style. It beats writing an equivalent series of `new FormControl(...)` statements.

{@a validators}

Validators.required

Though this guide doesn't go deeply into validations, here is one example that demonstrates the simplicity of using `Validators.required` in reactive forms.

First, import the `Validators` symbol.

To make the `name` `FormControl` required, replace the `name` property in the `FormGroup` with an array. The first item is the initial value for `name`; the second is the required validator, `Validators.required`.

Reactive validators are simple, composable functions. Configuring validation is harder in template-driven forms where you must wrap validators in a directive.

Update the diagnostic message at the bottom of the template to display the form's validity status.

The browser displays the following:

`Validators.required` is working. The status is `INVALID` because the input box has no value. Type into the input box to see the status change from `INVALID` to `VALID`.

In a real app, you'd replace the diagnosic message with a user-friendly experience.

Using `Validators.required` is optional for the rest of the guide. It remains in each of the following examples with the same configuration.

For more on validating Angular forms, see the Form Validation guide.

More FormControls

A hero has more than a name. A hero has an address, a super power and sometimes a sidekick too.

The address has a state property. The user will select a state with a `<select>` box and you'll populate the `<option>` elements with states. So import `states` from `data-model.ts`.

Declare the `states` property and add some address `FormControls` to the `heroForm` as follows.

Then add corresponding markup in `hero-detail.component.html` within the `form` element.

Reminder: Ignore the many mentions of `form-group`, `form-control`, `center-block`, and `checkbox` in this markup. Those are _bootstrap_ CSS classes that Angular itself ignores. Pay attention to the `formGroupName` and `formControlName` attributes. They are the Angular directives that bind the HTML controls to the Angular `FormGroup` and `FormControl` properties in the component class.

The revised template includes more text inputs, a select box for the `state`, radio buttons for the `power`, and a checkbox for the `sidekick`.

You must bind the option's value property with `[value]="state"`. If you do not bind the value, the select shows the first option from the data model.

The component *class* defines control properties without regard for their representation in the template. You define the `state`, `power`, and `sidekick` controls the same way you defined the `name` control. You tie these controls to the template HTML elements in the same way, specifying the `FormControl` name with the `formControlName` directive.

See the API reference for more information about radio buttons, selects, and checkboxes.

{@a grouping}

Nested FormGroups

This form is getting big and unwieldy. You can group some of the related `FormControls` into a nested `FormGroup`. The `street`, `city`, `state`, and `zip` are properties that would make a good *address* `FormGroup`. Nesting groups and controls in this way allows you to mirror the hierarchical structure of the data model and helps track validation and state for related sets of controls.

You used the `FormBuilder` to create one `FormGroup` in this component called `heroForm`. Let that be the parent `FormGroup`. Use `FormBuilder` again to create a child `FormGroup` that encapsulates the address controls; assign the result to a new `address` property of the parent `FormGroup`.

You've changed the structure of the form controls in the component class; you must make corresponding adjustments to the component template.

In `hero-detail.component.html`, wrap the address-related `FormControls` in a `div`. Add a `formGroupName` directive to the `div` and bind it to `"address"`. That's the property of the *address* child `FormGroup` within the parent `FormGroup` called `heroForm`.

To make this change visually obvious, slip in an `<h4>` header near the top with the text, *Secret Lair*. The new *address* HTML looks like this:

After these changes, the JSON output in the browser shows the revised *form model* with the nested address `FormGroup`:

```
heroForm value: { "name": "", "address": { "street": "", "city": "",
"state": "", "zip": "" } }
```

Great! You've made a group and you can see that the template and the form model are talking to one another.

{@a properties}

Inspect *FormControl* Properties

At the moment, you're dumping the entire form model onto the page. Sometimes you're interested only in the state of one particular `FormControl`.

You can inspect an individual `FormControl` within a form by extracting it with the `.get()` method. You can do this *within* the component class or display it on the page by adding the following to the template, immediately after the `{{form.value | json}}` interpolation as follows:

To get the state of a `FormControl` that's inside a `FormGroup`, use dot notation to path to the control.

You can use this technique to display *any* property of a `FormControl` such as one of the following:

Property	Description
`myControl.value`	the value of a `FormControl`.
`myControl.status`	the validity of a `FormControl`. Possible values: `VALID`, `INVALID`, `PENDING`, or `DISABLED`.
`myControl.pristine`	`true` if the user has _not_ changed the value in the UI. Its opposite is `myControl.dirty`.
`myControl.untouched`	`true` if the control user has not yet entered the HTML control and triggered its blur event. Its opposite is `myControl.touched`.

Learn about other `FormControl` properties in the *AbstractControl* API reference.

One common reason for inspecting `FormControl` properties is to make sure the user entered valid values. Read more about validating Angular forms in the Form Validation guide.

{@a data-model-form-model}

The *data model* and the *form model*

At the moment, the form is displaying empty values. The `HeroDetailComponent` should display values of a hero, possibly a hero retrieved from a remote server.

In this app, the `HeroDetailComponent` gets its hero from a parent `HeroListComponent`

The `hero` from the server is the ***data model***. The `FormControl` structure is the ***form model***.

The component must copy the hero values in the *data model* into the *form model*. There are two important implications:

1. The developer must understand how the properties of the *data model* map to the properties of the *form model*.

2. User changes flow from the DOM elements to the *form model*, not to the *data model*. The form controls never update the *data model*.

The *form* and *data* model structures need not match exactly. You often present a subset of the *data model* on a particular screen. But it makes things easier if the shape of the *form model* is close to the shape of the *data model*.

In this `HeroDetailComponent`, the two models are quite close.

Recall the definition of `Hero` in `data-model.ts`:

Here, again, is the component's `FormGroup` definition.

There are two significant differences between these models:

1. The `Hero` has an `id`. The form model does not because you generally don't show primary keys to users.
2. The `Hero` has an array of addresses. This form model presents only one address, a choice revisited below.

Nonetheless, the two models are pretty close in shape and you'll see in a moment how this alignment facilitates copying the *data model* properties to the *form model* with the `patchValue` and `setValue` methods.

Take a moment to refactor the *address* `FormGroup` definition for brevity and clarity as follows:

Also be sure to update the import from `data-model` so you can reference the `Hero` and `Address` classes:

{@a set-data}

Populate the form model with *setValue* and *patchValue*

Previously you created a control and initialized its value at the same time. You can also initialize or reset the values *later* with the `setValue` and `patchValue` methods.

setValue

With **`setValue`**, you assign *every* form control value *at once* by passing in a data object whose properties exactly match the *form model* behind the `FormGroup`.

The `setValue` method checks the data object thoroughly before assigning any form control values.

It will not accept a data object that doesn't match the FormGroup structure or is missing values for any control in the group. This way, it can return helpful error messages if you have a typo or if you've nested controls incorrectly. `patchValue` will fail silently.

On the other hand, `setValue` will catch the error and report it clearly.

Notice that you can *almost* use the entire `hero` as the argument to `setValue` because its shape is similar to the component's `FormGroup` structure.

You can only show the hero's first address and you must account for the possibility that the `hero` has no addresses at all. This explains the conditional setting of the `address` property in the data object argument:

patchValue

With **`patchValue`**, you can assign values to specific controls in a `FormGroup` by supplying an object of key/value pairs for just the controls of interest.

This example sets only the form's `name` control.

With **`patchValue`** you have more flexibility to cope with wildly divergent data and form models. But unlike `setValue`, `patchValue` cannot check for missing control values and does not throw helpful errors.

When to set form model values (*ngOnChanges*)

Now you know *how* to set the *form model* values. But *when* do you set them? The answer depends upon when the component gets the *data model* values.

The `HeroDetailComponent` in this reactive forms sample is nested within a *master/detail* `HeroListComponent` (discussed below). The `HeroListComponent` displays hero names to the user. When the user clicks on a hero, the list component passes the selected hero into the `HeroDetailComponent` by binding to its `hero` input property.

In this approach, the value of `hero` in the `HeroDetailComponent` changes every time the user selects a new hero. You should call *setValue* in the ngOnChanges hook, which Angular calls whenever the input `hero` property changes as the following steps demonstrate.

First, import the `OnChanges` and `Input` symbols in `hero-detail.component.ts`.

Add the `hero` input property.

Add the `ngOnChanges` method to the class as follows:

reset the form flags

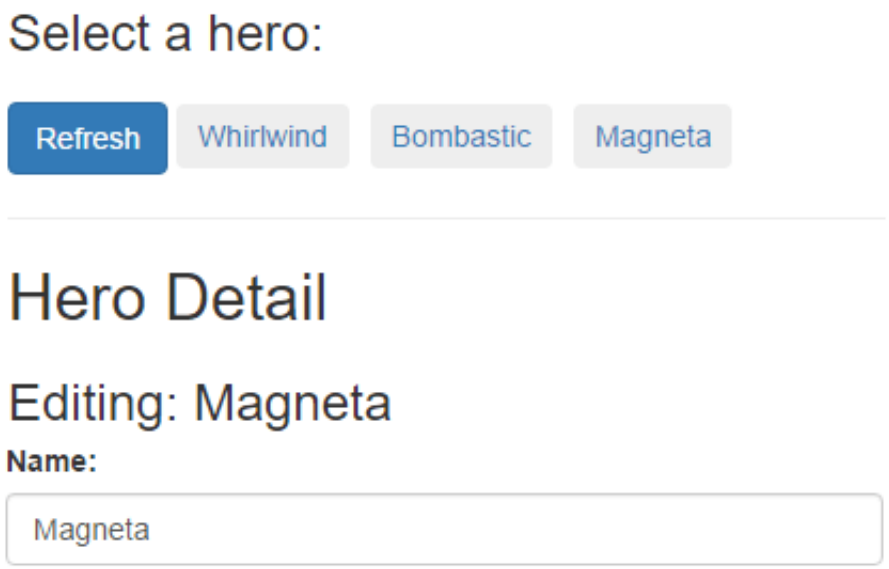
You should reset the form when the hero changes so that control values from the previous hero are cleared and status flags are restored to the *pristine* state. You could call `reset` at the top of `ngOnChanges` like this.

The `reset` method has an optional `state` value so you can reset the flags *and* the control values at the same time. Internally, `reset` passes the argument to `setValue`. A little refactoring and `ngOnChanges` becomes this:

{@a hero-list}

Create the *HeroListComponent* and *HeroService*

The `HeroDetailComponent` is a nested sub-component of the `HeroListComponent` in a *master/detail* view. Together they look a bit like this:



The `HeroListComponent` uses an injected `HeroService` to retrieve heroes from the server and then presents those heroes to the user as a series of buttons. The `HeroService` emulates an HTTP service. It returns an `Observable` of heroes that resolves after a short delay, both to simulate network latency and to indicate visually the necessarily asynchronous nature of the application.

When the user clicks on a hero, the component sets its `selectedHero` property which is bound to the `hero` input property of the `HeroDetailComponent`. The `HeroDetailComponent` detects the changed hero and re-sets its form with that hero's data values.

A "Refresh" button clears the hero list and the current selected hero before refetching the heroes.

The remaining `HeroListComponent` and `HeroService` implementation details are not relevant to understanding reactive forms. The techniques involved are covered elsewhere in the documentation, including the *Tour of Heroes* here and here.

If you're coding along with the steps in this reactive forms tutorial, generate the pertinent files based on the source code displayed below. Notice that `hero-list.component.ts` imports `Observable` and `finally` while `hero.service.ts` imports `Observable`, `of`, and `delay` from `rxjs`. Then return here to learn about *form array* properties.

{@a form-array}

Use *FormArray* to present an array of *FormGroups*

So far, you've seen `FormControls` and `FormGroups`. A `FormGroup` is a named object whose property values are `FormControls` and other `FormGroups`.

Sometimes you need to present an arbitrary number of controls or groups. For example, a hero may have zero, one, or any number of addresses.

The `Hero.addresses` property is an array of `Address` instances. An *address* `FormGroup` can display one `Address`. An Angular `FormArray` can display an array of *address* `FormGroups`.

To get access to the `FormArray` class, import it into `hero-detail.component.ts`:

To *work* with a `FormArray` you do the following:

1. Define the items (`FormControls` or `FormGroups`) in the array.
2. Initialize the array with items created from data in the *data model*.
3. Add and remove items as the user requires.

In this guide, you define a `FormArray` for `Hero.addresses` and let the user add or modify addresses (removing addresses is your homework).

You'll need to redefine the form model in the `HeroDetailComponent` constructor, which currently only displays the first hero address in an *address* `FormGroup`.

From *address* to *secret lairs*

From the user's point of view, heroes don't have *addresses*. *Addresses* are for mere mortals. Heroes have *secret lairs*! Replace the *address* `FormGroup` definition with a *secretLairs* `FormArray` definition:

Changing the form control name from `address` to `secretLairs` drives home an important point: the _form model_ doesn't have to match the _data model_. Obviously there has to be a relationship between the two. But it can be anything that makes sense within the application domain. _Presentation_ requirements often differ from _data_ requirements. The reactive forms approach both emphasizes and facilitates this distinction.

Initialize the "secretLairs" *FormArray*

The default form displays a nameless hero with no addresses.

You need a method to populate (or repopulate) the *secretLairs* with actual hero addresses whenever the parent

`HeroListComponent` sets the `HeroDetailComponent.hero` input property to a new `Hero`.

The following `setAddresses` method replaces the *secretLairs* `FormArray` with a new `FormArray`, initialized by an array of hero address `FormGroups`.

Notice that you replace the previous `FormArray` with the **`FormGroup.setControl` method**, not with `setValue`. You're replacing a *control*, not the *value* of a control.

Notice also that the *secretLairs* `FormArray` contains **`FormGroups`**, not `Addresses`.

Get the *FormArray*

The `HeroDetailComponent` should be able to display, add, and remove items from the *secretLairs* `FormArray`.

Use the `FormGroup.get` method to acquire a reference to that `FormArray`. Wrap the expression in a `secretLairs` convenience property for clarity and re-use.

Display the *FormArray*

The current HTML template displays a single *address* `FormGroup`. Revise it to display zero, one, or more of the hero's *address* `FormGroups`.

This is mostly a matter of wrapping the previous template HTML for an address in a `<div>` and repeating that `<div>` with `*ngFor`.

The trick lies in knowing how to write the `*ngFor`. There are three key points:

1. Add another wrapping `<div>`, around the `<div>` with `*ngFor`, and set its `formArrayName` directive to `"secretLairs"`. This step establishes the *secretLairs* `FormArray` as the context for form controls in the inner, repeated HTML template.

2. The source of the repeated items is the `FormArray.controls`, not the `FormArray` itself. Each control is an *address* `FormGroup`, exactly what the previous (now repeated) template HTML expected.

3. Each repeated `FormGroup` needs a unique `formGroupName` which must be the index of the `FormGroup` in the `FormArray`. You'll re-use that index to compose a unique label for each address.

Here's the skeleton for the *secret lairs* section of the HTML template:

Here's the complete template for the *secret lairs* section:

Add a new lair to the *FormArray*

Add an `addLair` method that gets the *secretLairs* `FormArray` and appends a new *address* `FormGroup` to it.

Place a button on the form so the user can add a new *secret lair* and wire it to the component's `addLair` method.

Be sure to **add the `type="button"` attribute**. In fact, you should always specify a button's `type`. Without an explicit type, the button type defaults to "submit". When you later add a _form submit_ action, every "submit" button triggers the submit action which might do something like save the current changes. You do not want to save changes when the user clicks the _Add a Secret Lair_ button.

Try it!

Back in the browser, select the hero named "Magneta". "Magneta" doesn't have an address, as you can see in the diagnostic JSON at the bottom of the form.

```
heroForm value: { "name": "Magneta", "secretLairs": [] }
```

Click the "*Add a Secret Lair*" button. A new address section appears. Well done!

Remove a lair

This example can *add* addresses but it can't *remove* them. For extra credit, write a `removeLair` method and wire it to a button on the repeating address HTML.

{@a observe-control}

Observe control changes

Angular calls `ngOnChanges` when the user picks a hero in the parent `HeroListComponent`. Picking a hero changes the `HeroDetailComponent.hero` input property.

Angular does *not* call `ngOnChanges` when the user modifies the hero's *name* or *secret lairs*. Fortunately, you can learn about such changes by subscribing to one of the form control properties that raises a change event.

These are properties, such as `valueChanges`, that return an RxJS `Observable`. You don't need to know much about RxJS `Observable` to monitor form control values.

Add the following method to log changes to the value of the *name* `FormControl`.

Call it in the constructor, after creating the form.

The `logNameChange` method pushes name-change values into a `nameChangeLog` array. Display that array at the bottom of the component template with this `*ngFor` binding:

Return to the browser, select a hero (e.g, "Magneta"), and start typing in the *name* input box. You should see a new name in the log after each keystroke.

When to use it

An interpolation binding is the easier way to *display* a name change. Subscribing to an observable form control property is handy for triggering application logic *within* the component class.

{@a save}

Save form data

The `HeroDetailComponent` captures user input but it doesn't do anything with it. In a real app, you'd probably save those hero changes. In a real app, you'd also be able to revert unsaved changes and resume editing. After you implement both features in this section, the form will look like this:

Save

In this sample application, when the user submits the form, the `HeroDetailComponent` will pass an

instance of the hero *data model* to a save method on the injected `HeroService`.

This original `hero` had the pre-save values. The user's changes are still in the *form model*. So you create a new `hero` from a combination of original hero values (the `hero.id`) and deep copies of the changed form model values, using the `prepareSaveHero` helper.

Address deep copy Had you assigned the `formModel.secretLairs` to `saveHero.addresses` (see line commented out), the addresses in `saveHero.addresses` array would be the same objects as the lairs in the `formModel.secretLairs`. A user's subsequent changes to a lair street would mutate an address street in the `saveHero`. The `prepareSaveHero` method makes copies of the form model's `secretLairs` objects so that can't happen.

Revert (cancel changes)

The user cancels changes and reverts the form to the original state by pressing the *Revert* button.

Reverting is easy. Simply re-execute the `ngOnChanges` method that built the *form model* from the original, unchanged `hero` *data model*.

Buttons

Add the "Save" and "Revert" buttons near the top of the component's template:

The buttons are disabled until the user "dirties" the form by changing a value in any of its form controls (`heroForm.dirty`).

Clicking a button of type `"submit"` triggers the `ngSubmit` event which calls the component's `onSubmit` method. Clicking the revert button triggers a call to the component's `revert` method. Users now can save or revert changes.

This is the final step in the demo. Try the .

Summary

- How to create a reactive form component and its corresponding template.
- How to use `FormBuilder` to simplify coding a reactive form.
- Grouping `FormControls`.
- Inspecting `FormControl` properties.
- Setting data with `patchValue` and `setValue`.
- Adding groups dynamically with `FormArray`.
- Observing changes to the value of a `FormControl`.

- Saving form changes.

{@a source-code}

The key files of the final version are as follows:

You can download the complete source for all steps in this guide from the Reactive Forms Demo live example.