

FORMS

Developer Preview Only - some details may change

A form creates a cohesive, effective, and compelling data entry experience. An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.

We've all used a form to login, submit a help request, place an order, book a flight, schedule a meeting and perform countless other data entry tasks. Forms are the mainstay of business applications.

Any seasoned web developer can slap together an HTML form with all the right tags. It's more challenging to create a cohesive data entry experience that guides the user efficiently and effectively through the workflow behind the form.

That takes design skills that are, to be frank, well out of scope for this chapter.

It also takes framework support for **two-way data binding, change tracking, validation, and error handling** ... which we shall cover in this chapter on Angular forms.

We will build a simple form from scratch, one step at a time. Along the way we'll learn

- How to build an Angular form with a component and template
- The `ngModel` two-way data binding syntax for reading and writing values to input controls
- The `ngControl` directive to track the change state and validity of form controls
- The special CSS classes that `ngControl` adds to form controls and how we can use them to provide strong visual feedback
- How to display validation errors to users and enable/disable form controls
- How to share information across controls with template local variables

[Live Example](#)

Template-Driven Forms

Many of us will build forms by writing templates in the Angular [template syntax](#) with the form-specific directives and techniques described in this chapter.

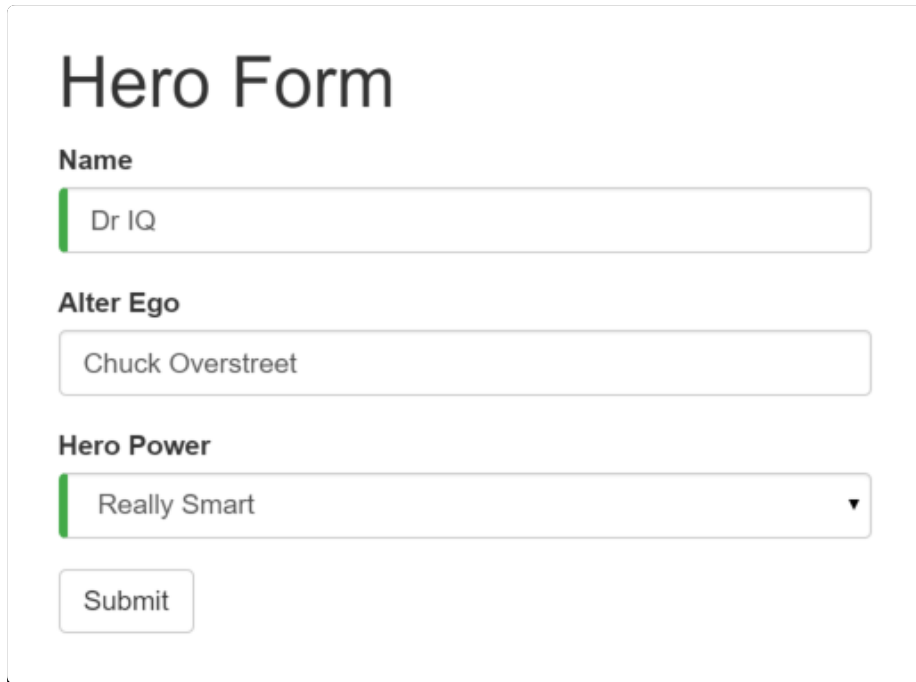
That's not the only way to create a form but it's the way we'll cover in this chapter.

We can build almost any form we need with an Angular template — login forms, contact forms ... pretty much any business forms. We can lay out the controls creatively, bind them to data, specify validation rules and display validation errors,

conditionally enable or disable specific controls, trigger built-in visual feedback, and much more.

It will be pretty easy because Angular handles many of the repetitive, boiler plate tasks we'd otherwise wrestle with ourselves.

We'll discuss and learn to build the following template-driven form:



Hero Form

Name
Dr IQ

Alter Ego
Chuck Overstreet

Hero Power
Really Smart ▼

Submit

Here at the *Hero Employment Agency* we use this form to maintain personal information about the heroes in our stable. Every hero needs a job. It's our company mission to match the right hero with the right crisis!

Two of the three fields on this form are required. Required fields have a green bar on the left to make them easy to spot.

If we delete the hero name, the form displays a validation error in an attention grabbing style:

Hero Form

Name

Name is required

Alter Ego

Hero Power

Submit

Note that the submit button is disabled and the "required" bar to the left of the input control changed from green to red.

We'll customize the colors and location of the "required" bar with standard CSS.

We will build this form in the following sequence of small steps

1. Create the `Hero` model class
2. Create the component that controls the form
3. Create a template with the initial form layout

4. Add the **ngModel** directive to each form input control
5. Add the **ngControl** directive to each form input control
6. Add custom CSS to provide visual feedback
7. Show and hide validation error messages
8. Handle form submission with **ngSubmit**
9. Disable the form's submit button until the form is valid

Setup

Create a new project folder (`angular2-forms`) and follow the steps in the [QuickStart](#).

Create the Hero Model Class

As users enter form data, we capture their changes and update an instance of a model. We can't layout the form until we know what the model looks like.

A model can be as simple as a "property bag" that holds facts about a thing of application importance. That describes well our `Hero` class with its three required fields (`id` , `name` , `power`) and one optional field (`alterEgo`).

Create a new file in the app folder called `hero.ts` and give it the following class definition:

app/hero.ts

```
1. export class Hero {  
2.  
3.   constructor(  
4.     public id: number,
```



```
5.     public name: string,  
6.     public power: string,  
7.     public alterEgo?: string  
8.   ) { }  
9.  
10. }
```

It's an anemic model with few requirements and no behavior. Perfect for our demo.

The TypeScript compiler generates a public field for each `public` constructor parameter and assigns the parameter's value to that field automatically when we create new heroes.

The `alterEgo` is optional and the constructor lets us omit it; note the `(?)` in `alterEgo?`.

We can create a new hero like this:

```
let myHero = new Hero(42, 'SkyDog',  
                      'Fetch any object at any distance', 'Leslie Rollover');  
console.log('My hero is called ' + myHero.name); // "My hero is called SkyDog"
```



Create a Form component

An Angular form has two parts: an HTML-based template and a code-based Component to handle data and user interactions.

We begin with the Component because it states, in brief, what the Hero editor can do.

Create a new file called `hero-form.component.ts` and give it the following definition:

app/hero-form.component.ts

```
1. import {Component} from 'angular2/core';
2. import {NgForm}     from 'angular2/common';
3. import { Hero }     from './hero';
4.
5. @Component({
6.   selector: 'hero-form',
7.   templateUrl: 'app/hero-form.component.html'
8. })
9. export class HeroFormComponent {
10.
11.   powers = ['Really Smart', 'Super Flexible',
12.             'Super Hot', 'Weather Changer'];
13.
14.   model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
15.
16.   submitted = false;
17.
18.   onSubmit() { this.submitted = true; }
19.
20.   // TODO: Remove this when we're done
21.   get diagnostic() { return JSON.stringify(this.model); }
22. }
```



There's nothing special about this component, nothing form-specific, nothing to distinguish it from any component we've written before.

Understanding this component requires only the Angular 2 concepts we've learned in previous chapters

1. We import the `Component` decorator from the Angular library as we usually do.
2. The `@Component` selector value of "hero-form" means we can drop this form in a parent template with a `<hero-form>` tag.
3. The `templateUrl` property points to a separate file for template HTML called `hero-form.component.html`.
4. We defined dummy data for `model` and `powers` as befits a demo. Down the road, we can inject a data service to get and save real data or perhaps expose these properties as [inputs and outputs](#) for binding to a parent component. None of this concerns us now and these future changes won't affect our form.
5. We threw in a `diagnostic` property at the end to return a JSON representation of our model. It'll help us see what we're doing during our development; we've left ourselves a cleanup note to discard it later.

Why don't we write the template inline in the component file as we often do elsewhere in the Developer Guide?

There is no "right" answer for all occasions. We like inline templates when they are short. Most form templates won't be short. TypeScript and JavaScript files generally aren't the best place to write (or read) large stretches of HTML and few editors are much help with files that have a mix of HTML and code. We also like short files with a clear and obvious purpose like this one.

We made a good choice to put the HTML template elsewhere. We'll write that template in a moment. Before we do, we'll take

a step back and revise the `app.component.ts` to make use of our new `HeroFormComponent`.

Revise the `app.component.ts`

`app.component.ts` is the application's root component. It will host our new `HeroFormComponent`.

Replace the contents of the "QuickStart" version with the following:

app/app.component.ts

```
1. import {Component}           from 'angular2/core';
2. import {HeroFormComponent} from './hero-form.component'
3.
4. @Component({
5.   selector: 'my-app',
6.   template: '<hero-form></hero-form>',
7.   directives: [HeroFormComponent]
8. })
9. export class AppComponent { }
```



There are only three changes:

1. We import the new `HeroFormComponent`.
2. The `template` is simply the new element tag identified by the component's `selector` property.

3. The `directives` array tells Angular that our template depends upon the `HeroFormComponent` which is itself a Directive (as are all Components).

Create an initial HTML Form Template

Create a new template file called `hero-form.component.html` and give it the following definition:

app/hero-form.component.html

```
1. <div class="container">
2.   <h1>Hero Form</h1>
3.   <form>
4.     <div class="form-group">
5.       <label for="name">Name</label>
6.       <input type="text" class="form-control" required>
7.     </div>
8.
9.     <div class="form-group">
10.      <label for="alterEgo">Alter Ego</label>
11.      <input type="text" class="form-control">
12.    </div>
13.
14.    <button type="submit" class="btn btn-default">Submit</button>
15.
16.  </form>
17. </div>
```



That is plain old HTML 5. We're presenting two of the `Hero` fields, `name` and `alterEgo`, and opening them up for user input in input boxes.

The `Name` `<input>` control has the HTML5 `required` attribute; the `Alter Ego` `<input>` control does not because `alterEgo` is optional.

We've got a `Submit` button at the bottom with some classes on it.

We are not using Angular yet. There are no bindings. No extra directives. Just layout.

The `container`, `form-group`, `form-control`, and `btn` classes come from [Twitter Bootstrap](#). Purely cosmetic. We're using Bootstrap to gussy up our form. Hey, what's a form without a little style!

ANGULAR FORMS DO NOT REQUIRE A STYLE LIBRARY

Angular makes no use of the `container`, `form-group`, `form-control`, and `btn` classes or the styles of any external library. Angular apps can use any CSS library ... or none at all.

Let's add the stylesheet.

1. Open a terminal window in the application root folder and enter the command:

```
npm install bootstrap --save
```



2. Open `index.html` and add the following link to the `<head>`.

```
<link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.min.css">
```



Add Powers with ***ngFor**

Our hero may choose one super power from a fixed list of Agency-approved powers. We maintain that list internally (in `HeroFormComponent`).

We'll add a `select` to our form and bind the options to the `powers` list using `NgFor`, a technique we might have seen before in the [Displaying Data](#) chapter.

Add the following HTML *immediately below* the *Alter Ego* group.

app/hero-form.component.html (excerpt)

```
<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control" required>
    <option *ngFor="#p of powers" [value]="p">{{p}}</option>
  </select>
</div>
```

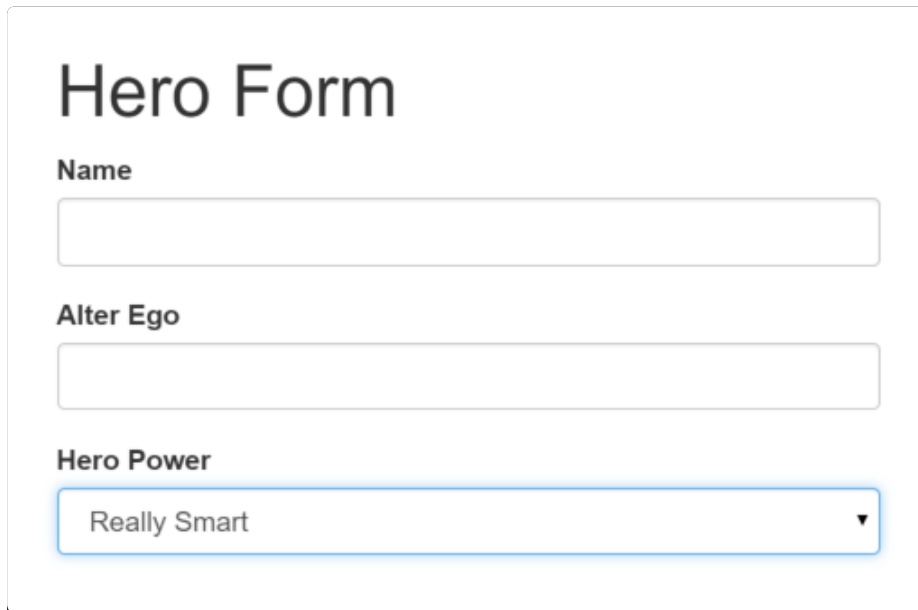


We are repeating the `<options>` tag for each power in the list of Powers. The `#p` local template variable is a different

power in each iteration; we display its name using the interpolation syntax with the double-curly-braces.

Two-way data binding with `*ngModel`

Running the app right now would be disappointing.



The screenshot shows a web form titled "Hero Form". It contains three input fields: "Name" (a text input), "Alter Ego" (a text input), and "Hero Power" (a dropdown menu). The "Hero Power" dropdown is currently showing "Really Smart" and has a blue border.

We don't see hero data because we are not binding to the `Hero` yet. We know how to do that from earlier chapters.

[Displaying Data](#) taught us Property Binding. [User Input](#) showed us how to listen for DOM events with an Event Binding and how to update a component property with the displayed value.

Now we need to display, listen, and extract at the same time.

We could use those techniques again in our form. Instead we'll introduce something new, the `NgModel` directive, that makes

binding our form to the model super-easy.

Find the `<input>` tag for the "Name" and update it like this

app/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" required  
  [(ngModel)]="model.name" >  
  TODO: remove this: {{model.name}}
```



We appended a diagnostic interpolation after the input tag so we can see what we're doing. We left ourselves a note to throw it away when we're done.

Focus on the binding syntax: `[(ngModel)]="..."`.

If we ran the app right now and started typing in the *Name* input box, adding and deleting characters, we'd see them appearing and disappearing from the interpolated text. At some point it might look like this.

Dr IQ 3000

TODO: remove this: Dr IQ 3000

The diagnostic is evidence that we really are flowing values from the input box to the model and back again. **That's two-way data binding!**

Let's add similar `[(ngModel)]` bindings to *Alter Ego* and *Hero Power*. We'll ditch the input box binding message and add a new binding at the top to the component's `diagnostic` property. Then we can confirm that two-way data binding works for the entire *Hero model*.

After revision the core of our form should have three `[(ngModel)]` bindings that look much like this:

app/hero-form.component.html (excerpt)

```
1. {{diagnostic}}
2. <div class="form-group">
3.   <label for="name">Name</label>
4.   <input type="text" class="form-control" required
5.     [(ngModel)]="model.name" >
6. </div>
7.
8. <div class="form-group">
9.   <label for="alterEgo">Alter Ego</label>
10.  <input type="text" class="form-control"
11.    [(ngModel)]="model.alterEgo">
12. </div>
13.
14. <div class="form-group">
15.   <label for="power">Hero Power</label>
16.   <select class="form-control" required
17.     [(ngModel)]="model.power" >
18.     <option *ngFor="#p of powers" [value]="p">{{p}}</option>
19.   </select>
20. </div>
```



If we ran the app right now and changed every Hero model property, the form might display like this:

Hero Form

```
{ "id": 18, "name": "Dr IQ 3000", "power": "Super Flexible", "alterEgo": "Chuck OverUnderStreet" }
```

Name

Alter Ego

Hero Power

The diagnostic near the top of the form confirms that all of our changes are reflected in the model.

Delete the `{{diagnostic}}` binding at the top as it has served its purpose.

Inside [(ngModel)]

This section is an optional deep dive into [(ngModel)]. Not interested? Skip ahead!

The punctuation in the binding syntax, `[()]`, is a good clue to what's going on.

In a Property Binding, a value flows from the model to a target property on screen. We identify that target property by surrounding its name in brackets, `[]`. This is a one-way data binding **from the model to the view**.

In an Event Binding, we flow the value from the target property on screen to the model. We identify that target property by surrounding its name in parentheses, `()`. This is a one-way data binding in the opposite direction **from the view to the model**.

No wonder Angular chose to combine the punctuation as `[()]` to signify a two-way data binding and a **flow of data in both directions**.

In fact, we can break the `NgModel` binding into its two separate modes as we do in this re-write of the "Name" `<input>` binding:

app/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" required
  [ngModel]="model.name"
  (ngModelChange)="model.name = $event" >
  TODO: remove this: {{model.name}}
```



The Property Binding should feel familiar. The Event Binding might seem strange.

The `ngModelChange` is not an `<input>` element event. It is actually an event property of the `NgModel` directive. When Angular sees a binding target in the form `[(x)]`, it expects the `x` directive to have an `x` input property and an `xChange` output property.

The other oddity is the template expression, `model.name = $event`. We're used to seeing an `$event` object coming from a DOM event. The `ngModelChange` property doesn't produce a DOM event; it's an Angular `EventEmitter` property that returns the input box value when it fires — which is precisely what we should assign to the model's `name` property.

Nice to know but is it practical? We almost always prefer `[(ngModel)]`. We might split the binding if we had to do something special in the event handling such as debounce or throttle the key strokes.

Learn more about `NgModel` and other template syntax in the [Template Syntax](#) chapter.

Track change-state and validity with **ngControl**

A form isn't just about data binding. We'd also like to know the state of the controls on our form. The `NgControl` directive keeps track of control state for us.

NGCONTROL REQUIRES FORM

The `NgControl` is one of a family of `NgForm` directives that can only be applied to a control within a `<form>` tag.

Our application can ask an `NgControl` if the user touched the control, if the value changed, or if the value became invalid.

`NgControl` doesn't just track state; it updates the control with special Angular CSS classes from the set we listed above. We can leverage those class names to change the appearance of the control and make messages appear or disappear.

We'll explore those effects soon. Right now we should **add `ngControl` to all three form controls**, starting with the *Name* input box

app/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" required  
  [(ngModel)]="model.name"  
  ngControl="name" >
```



Be sure to assign a unique name to each `ngControl` directive.

Angular registers controls under their `ngControl` names with the `NgForm`. We didn't add the `NgForm` directive explicitly but it's here and we'll talk about it [later in this chapter](#).

Add Custom CSS for Visual Feedback

`NgControl` doesn't just track state. It updates the control with three classes that reflect the state.

State	Class if true	Class if false
Control has been visited	<code>ng-touched</code>	<code>ng-untouched</code>
Control's value has changed	<code>ng-dirty</code>	<code>ng-pristine</code>
Control's value is valid	<code>ng-valid</code>	<code>ng-invalid</code>

Let's add a temporary [local template variable](#) named **spy** to the "Name" `<input>` tag and use the spy to display those classes.

app/hero-form.component.html (excerpt)

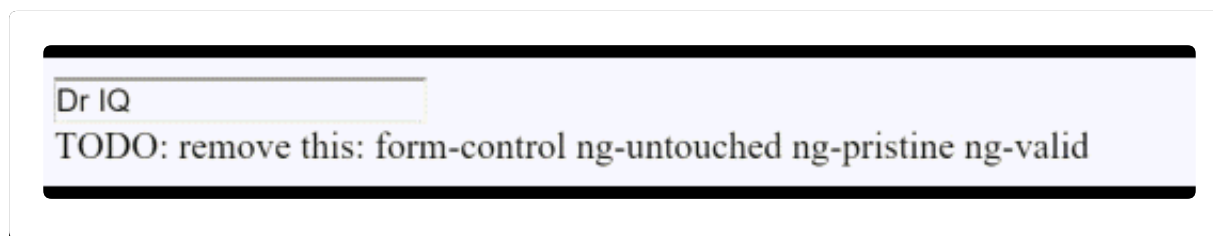
```
<input type="text" class="form-control" required  
  [(ngModel)]="model.name"  
  ngControl="name" #spy >  
<br>TODO: remove this: {{spy.className}}
```



Now run the app and focus on the *Name* input box. Follow the next four steps *precisely*

1. Look but don't touched
2. Click in the input box, then click outside the text input box
3. Add slashes to the end of the name
4. Erase the name

The actions and effects are as follows:



We should be able to see the following four sets of class names and their transitions:

The diagram illustrates four input fields with their corresponding AngularJS classes and states:

- Untouched:** Input field contains "Dr IQ". The classes are `ng-untouched`, `ng-pristine`, and `ng-valid`.
- Touched:** Input field contains "Dr IQ". The classes are `ng-pristine`, `ng-valid`, and `ng-touched`.
- Changed:** Input field contains "Dr IQ////". The classes are `ng-valid`, `ng-touched`, and `ng-dirty`.
- Invalid:** Input field is empty. The classes are `ng-touched`, `ng-dirty`, and `ng-invalid`.

Each input field has a "TODO: remove this: form-control" label above it.

The (`ng-valid` | `ng-invalid`) pair are most interesting to us. We want to send a strong visual signal when the data are invalid and we want to mark required fields.

We realize we can do both at the same time with a colored bar on the left of the input box:

The diagram shows three input fields with colored bars on the left indicating their state:

- Valid + Required:** Input field contains "Dr IQ". A green bar is on the left.
- Valid + Optional:** Input field contains "Chuck Overstreet". A light green bar is on the left.
- Invalid (required | optional):** Input field is empty. A red bar is on the left.

We achieve this effect by adding two styles to a new `styles.css` file that we add to our project as a sibling to `index.html`.

`styles.css`

```
.ng-valid[required] {  
  border-left: 5px solid #42A948; /* green */  
}  
  
.ng-invalid {  
  border-left: 5px solid #a94442; /* red */  
}
```



These styles select for the two Angular validity classes and the HTML 5 "required" attribute.

We update the `<head>` of the `index.html` to include this style sheet.

index.html (excerpt)

```
<link rel="stylesheet" href="styles.css">
```

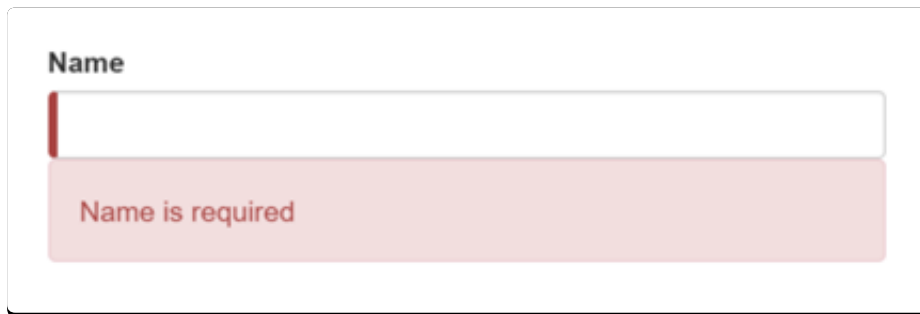


Show and Hide Validation Error messages

We can do better.

The "Name" input box is required. Clearing it turns the bar red. That says *something* is wrong but we don't know *what* is wrong or what to do about it. We can leverage the `ng-invalid` class to reveal a helpful message.

Here's the way it should look when the user deletes the name:



To achieve this effect we extend the `<input>` tag with

1. a [local template variable](#)
2. the "is required" message in a nearby `<div>` which we'll display only if the control is invalid.

Here's how we do it for the *name* input box:

app/hero-form.component.html (excerpt)

```
<label for="name">Name</label>
<input type="text" class="form-control" required
  [(ngModel)]="model.name"
  ngControl="name" #name="ngForm" >
<div [hidden]="name.valid || name.pristine" class="alert alert-danger">
  Name is required
</div>
```



When we added the `ngControl` directive, we bound it to the the model's `name` property.

Here we initialize a template local variable (`name`) with the value "ngForm" (`#name="ngForm"`). Angular recognizes that

syntax and re-sets the `name` local template variable to the `ngControl` directive instance. In other words, the `name` local template variable becomes a handle on the `ngControl` object for this input box.

Now we can control visibility of the "name" error message by binding properties of the `name` control to the message `<div>` element's `hidden` property.

app/hero-form.component.html (excerpt)

```
<div [hidden]="name.valid || name.pristine" class="alert alert-danger">
```



In this example, we hide the message when the control is valid or pristine; pristine means the user hasn't changed the value since it was displayed in this form.

This user experience is the developer's choice. Some folks want to see the message at all times. If we ignore the `pristine` state, we would hide the message only when the value is valid. If we arrive in this component with a new (blank) hero or an invalid hero, we'll see the error message immediately, before we've done anything.

Some folks find that behavior disconcerting. They only want to see the message when the user makes an invalid change. Hiding the message while the control is "pristine" achieves that goal. We'll see the significance of this choice when we [add a new hero](#) to the form.

The NgForm directive

We just set a template local variable with the value of an `NgForm` directive. Why did that work? We didn't add the `NgForm` directive explicitly.

Angular added it surreptitiously, wrapping it around the `<form>` element

The `NgForm` directive supplements the `form` element with additional features. It collects `Controls` (elements identified by an `ngControl` directive) and monitors their properties including their validity. It also has its own `valid` property which is true only if every contained control is valid.

The Hero *Alter Ego* is optional so we can leave that be.

Hero *Power* selection is required. We can add the same kind of error handling to the `<select>` if we want but it's not imperative because the selection box already constrains the power to valid value.

Add a hero and reset the form

We'd like to add a new hero in this form. We place a "New Hero" button at the bottom of the form and bind its click event to a component method.

app/hero-form.component.html (New Hero button)

```
<button type="button" class="btn btn-default" (click)="newHero()">New Hero</button>
```



app/hero-form.component.ts (New Hero method - v1)

```
newHero() {  
  this.model = new Hero(42, '', '');  
}
```



Run the application again, click the *New Hero* button, and the form clears. The *required* bars to the left of the input box are red, indicating invalid `name` and `power` properties. That's understandable as these are required fields. The error messages are hidden because the form is pristine; we haven't changed anything yet.

Enter a name and click *New Hero* again. This time we see an error message! Why? We don't want that when we display a new (empty) hero.

Inspecting the element in the browser tools reveals that the *name* input box is no longer pristine. Replacing the hero *did not* restore the *pristine state* of the control.

Upon reflection, we realize that Angular cannot distinguish between replacing the entire hero and clearing the `name` property programmatically. Angular makes no assumptions and leaves the control in its current, dirty state.

We'll have to reset the form controls manually with a small trick. We add an `active` flag to the component, initialized to `true`. When we add a new hero, we toggle `active` false and then immediately back to true with a quick `setTimeout`.

app/hero-form.component.ts (New Hero method - final)

```
active = true;

newHero() {
  this.model = new Hero(42, '', '');
  this.active = false;
  setTimeout(() => this.active=true, 0);
}
```



```
}
```

Then we bind the form element to this `active` flag.

app/hero-form.component.html (Form tag)

```
<form *ngIf="active">
```



With `NgIf` bound to the `active` flag, clicking "New Hero" removes the form from the DOM and recreates it in a blink of an eye. The re-created form is in a pristine state. The error message is hidden.

This is a temporary workaround while we await a proper form reset feature.

Submit the form with **ngSubmit**

The user should be able to submit this form after filling it in. The Submit button at the bottom of the form does nothing on its own but it will trigger a form submit because of its type (`type="submit"`).

A "form submit" is useless at the moment. To make it useful, we'll update the `<form>` tag with another Angular directive, `NgSubmit` , and bind it to the `HeroFormComponent.submit()` method with an event binding

```
<form *ngIf="active" (ngSubmit)="onSubmit()" #heroForm="ngForm">
```



We slipped in something extra there at the end! We defined a template local variable, `#heroForm`, and initialized it with the value, `ngForm`.

The variable `heroForm` is now a handle to the `NgForm` directive that we [discussed earlier](#). This time `heroForm` remains a reference to the form as a whole.

Later in the template we bind the button's `disabled` property to the form's over-all validity via the `heroForm` variable. Here's that bit of markup:

```
<button type="submit" class="btn btn-default" [disabled]="!heroForm.form.valid">Submit</button>
```



Re-run the application. The form opens in a valid state and the button is enabled.

Now delete the *Name*. We violate the "name required" rule which is duly noted in our error message as before. And now the Submit button is also disabled.

Not impressed? Think about it for a moment. What would we have to do to wire the button's enable/disabled state to the form's validity without Angular's help?

For us, it was as simple as

1. Define a template local variable on the (enhanced) form element
2. Reference that variable in a button some 50 lines away.

Toggle two form regions (extra credit)

Submitting the form isn't terribly dramatic at the moment.

An unsurprising observation for a demo. To be honest, jazzing it up won't teach us anything new about forms. But this is an opportunity to exercise some of our newly won binding skills. If you're not interested, you can skip to the chapter's conclusion and not miss a thing.

Let's do something more strikingly visual. Let's hide the data entry area and display something else.

Start by wrapping the form in a `<div>` and bind its `hidden` property to the `HeroFormComponent.submitted` property.

app/hero-form.component.html (excerpt)

```
<div [hidden]="submitted">
  <h1>Hero Form</h1>
  <form *ngIf="active" (ngSubmit)="onSubmit()" #heroForm="ngForm">

    <!-- ... all of the form ... -->

  </form>
</div>
```



The main form is visible from the start because the `submitted` property is false until we submit the form, as this

fragment from the `HeroFormComponent` reminds us:

```
submitted = false;

onSubmit() { this.submitted = true; }
```



When we click the Submit button, the `submitted` flag becomes true and the form disappears as planned.

Now we need to show something else while the form is in the submitted state. Add the following block of HTML below the `<div>` wrapper we just wrote:

app/hero-form.component.html (excerpt)

```
1. <div [hidden]="!submitted">
2.   <h2>You submitted the following:</h2>
3.   <div class="row">
4.     <div class="col-xs-3">Name</div>
5.     <div class="col-xs-9 pull-left">{{ model.name }}</div>
6.   </div>
7.   <div class="row">
8.     <div class="col-xs-3">Alter Ego</div>
9.     <div class="col-xs-9 pull-left">{{ model.alterEgo }}</div>
10.  </div>
11.  <div class="row">
12.    <div class="col-xs-3">Power</div>
13.    <div class="col-xs-9 pull-left">{{ model.power }}</div>
14.  </div>
15.  <br>
```



```
16.   <button class="btn btn-default" (click)="submitted=false">Edit</button>
17. </div>
```

There's our hero again, displayed read-only with interpolation bindings. This slug of HTML only appears while the component is in the submitted state.

We added an Edit button whose click event is bound to an expression that clears the `submitted` flag.

When we click it, this block disappears and the editable form reappears.

That's as much drama as we can muster for now.

Conclusion

The Angular 2 form discussed in this chapter takes advantage of the following framework features to provide support for data modification, validation and more:

- An Angular HTML form template.
- A form component class with a `Component` decorator.
- The `ngSubmit` directive for handling the form submission.
- Template local variables such as `#heroForm`, `#name`, `#alter-ego` and `#power`.
- The `ngModel` directive for two-way data binding.
- The `ngControl` for validation and form element change tracking.
- The local variable's `valid` property on input controls to check if a control is valid and show/hide error messages.

- Controlling the submit button's enabled state by binding to `NgForm` validity.
- Custom CSS classes that provide visual feedback to users about invalid controls.

Our final project folder structure should look like this:

```
angular2-forms
├── app
│   ├── app.component.ts
│   ├── hero.ts
│   ├── hero-form.component.html
│   ├── hero-form.component.ts
│   └── main.ts
├── node_modules ...
├── typings ...
├── index.html
├── package.json
├── tsconfig.json
└── typings.json
```

Here's the final version of the source:

```
1. import {Component} from 'angular2/core';
2. import {NgForm}    from 'angular2/common';
```




```
3. import { Hero }    from './hero';
4.
5. @Component({
6.   selector: 'hero-form',
7.   templateUrl: 'app/hero-form.component.html'
8. })
9. export class HeroFormComponent {
10.
11.   powers = ['Really Smart', 'Super Flexible',
12.            'Super Hot', 'Weather Changer'];
13.
14.   model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
15.
16.   submitted = false;
17.
18.   onSubmit() { this.submitted = true; }
19.
20.   // Reset the form with a new hero AND restore 'pristine' class state
21.   // by toggling 'active' flag which causes the form
22.   // to be removed/re-added in a tick via NgIf
23.   // TODO: Workaround until NgForm has a reset method (#6822)
24.   active = true;
25.
26.   newHero() {
27.     this.model = new Hero(42, '', '');
28.     this.active = false;
29.     setTimeout(()=> this.active=true, 0);
30.   }
31. }
```

Next Step[Dependency Injection](#)