Forms

Forms are the mainstay of business applications. You use forms to log in, submit a help request, place an order, book a flight, schedule a meeting, and perform countless other data-entry tasks.

In developing a form, it's important to create a data-entry experience that guides the user efficiently and effectively through the workflow.

Developing forms requires design skills (which are out of scope for this page), as well as framework support for *two-way data binding, change tracking, validation, and error handling*, which you'll learn about on this page.

This page shows you how to build a simple form from scratch. Along the way you'll learn how to:

- Build an Angular form with a component and template.
- Use `ngModel` to create two-way data bindings for reading and writing input-control values.
- Track state changes and the validity of form controls.
- Provide visual feedback using special CSS classes that track the state of the controls.
- Display validation errors to users and enable/disable form controls.
- Share information across HTML elements using template reference variables.

You can run the in Plunker and download the code from there.

{@a template-driven}

Template-driven forms

You can build forms by writing templates in the Angular template syntax with the form-specific directives and techniques described in this page.

You can also use a reactive (or model-driven) approach to build forms. However, this page focuses on template-driven forms.

You can build almost any form with an Angular template—login forms, contact forms, and pretty much any business form. You 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.

Angular makes the process easy by handling many of the repetitive, boilerplate tasks you'd otherwise wrestle with yourself.

You'll learn to build a template-driven form that looks like this:

The *Hero Employment Agency* uses this form to maintain personal information about heroes. Every hero needs a job. It's the 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 you delete the hero name, the form displays a validation error in an attention-grabbing style:

Note that the *Submit* button is disabled, and the "required" bar to the left of the input control changes from green to red.

You can customize the colors and location of the "required" bar with standard CSS.

You'll build this form in 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. Bind data properties to each form control using the `ngModel` two-way data-binding syntax.
5. Add a `name` attribute 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 named `angular-forms` :

ng new angular-forms

Create the Hero model class

As users enter form data, you'll capture their changes and update an instance of a model. You can't lay out the form until you 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 the `Hero` class with its three required fields (`id` , `name` , `power`) and one optional field (`alterEgo`).

Using the Angular CLI, generate a new class named `Hero` :

ng generate class Hero

With this content:

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

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

The `alterEgo` is optional, so the constructor lets you omit it; note the question mark (?) in `alterEgo?` .

You can create a new hero like this:

Create a form component

An Angular form has two parts: an HTML-based *template* and a component *class* to handle data and user interactions programmatically. Begin with the class because it states, in brief, what the hero editor can do.

Using the Angular CLI, generate a new component named `HeroForm` :

ng generate component HeroForm

With this content:

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

Understanding this component requires only the Angular concepts covered in previous pages.

- The code imports the Angular core library and the `Hero` model you just created.
- The `@Component` selector value of "hero-form" means you can drop this form in a parent template with a `<hero-form>` tag.
- The `templateUrl` property points to a separate file for the template HTML.
- You defined dummy data for `model` and `powers` , as befits a demo.

Down the road, you can inject a data service to get and save real data or perhaps expose these properties as inputs and outputs (see Input and output properties on the Template Syntax page) for binding to a parent

component. This is not a concern now and these future changes won't affect the form.

- You added a `diagnostic` property to return a JSON representation of the model. It'll help you see what you're doing during development; you've left yourself a cleanup note to discard it later.

Revise *app.module.ts*

`app.module.ts` defines the application's root module. In it you identify the external modules you'll use in the application and declare the components that belong to this module, such as the `HeroFormComponent`.

Because template-driven forms are in their own module, you need to add the `FormsModule` to the array of `imports` for the application module before you can use forms.

Update it with the following:

There are two changes: 1. You import `FormsModule`. 1. You add the `FormsModule` to the list of `imports` defined in the `@NgModule` decorator. This gives the application access to all of the template-driven forms features, including `ngModel`.
If a component, directive, or pipe belongs to a module in the `imports` array, _don't_ re-declare it in the `declarations` array. If you wrote it and it should belong to this module, _do_ declare it in the `declarations` array.

Revise *app.component.html*

`AppComponent` is the application's root component. It will host the new `HeroFormComponent`.

Replace the contents of its template with the following:

There are only two changes. The `template` is simply the new element tag identified by the component's `selector` property. This displays the hero form when the application component is loaded. Don't forget to remove the `name` field from the class body as well.

Create an initial HTML form template

Update the template file with the following contents:

The language is simply HTML5. You'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.

You added a *Submit* button at the bottom with some classes on it for styling.

You're not using Angular yet. There are no bindings or extra directives, just layout.

In template driven forms, if you've imported `FormsModule`, you don't have to do anything to the `
` tag in order to make use of `FormsModule`. Continue on to see how this works.

The `container`, `form-group`, `form-control`, and `btn` classes come from Twitter Bootstrap. These classes are purely cosmetic. Bootstrap gives the form a little style.

Angular forms don't 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.

To add the stylesheet, open `styles.css` and add the following import line at the top:

Add powers with **ngFor*

The hero must choose one superpower from a fixed list of agency-approved powers. You maintain that list internally (in `HeroFormComponent`).

You'll add a `select` to the form and bind the options to the `powers` list using `ngFor`, a technique seen previously in the Displaying Data page.

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

This code repeats the `<option>` tag for each power in the list of powers. The `pow` template input variable is a different power in each iteration; you display its name using the interpolation syntax.

{@a ngModel}

Two-way data binding with *ngModel*

Running the app right now would be disappointing.

You don't see hero data because you're not binding to the `Hero` yet. You know how to do that from earlier pages. Displaying Data teaches property binding. User Input shows how to listen for DOM events with an event binding and how to update a component property with the displayed value.

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

You could use the techniques you already know, but instead you'll use the new `[(ngModel)]` syntax, which makes binding the form to the model easy.

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

You added a diagnostic interpolation after the input tag so you can see what you're doing. You left yourself a note to throw it away when you're done.

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

You need one more addition to display the data. Declare a template variable for the form. Update the `<form>` tag with `#heroForm="ngForm"` as follows:

The variable `heroForm` is now a reference to the `NgForm` directive that governs the form as a whole.

{@a ngForm} ### The _NgForm_ directive What `NgForm` directive? You didn't add an [NgForm](api/forms/NgForm) directive. Angular did. Angular automatically creates and attaches an `NgForm` directive to the `` tag. The `NgForm` directive supplements the `form` element with additional features. It holds the controls you created for the elements with an `ngModel` directive and `name` attribute, and monitors their properties, including their validity. It also has its own `valid` property which is true only *if every contained control* is valid.

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

The diagnostic is evidence that values really are flowing from the input box to the model and back again.

That's *two-way data binding*. For more information, see [Two-way binding with NgModel](guide/template-syntax#ngModel) on the the [Template Syntax](guide/template-syntax) page.

Notice that you also added a `name` attribute to the `<input>` tag and set it to "name", which makes sense for the hero's name. Any unique value will do, but using a descriptive name is helpful. Defining a `name` attribute is a requirement when using `[(ngModel)]` in combination with a form.

Internally, Angular creates `FormControl` instances and registers them with an `NgForm` directive that Angular attached to the `` tag. Each `FormControl` is registered under the name you assigned to the `name` attribute. Read more in the previous section, [The NgForm directive](guide/forms#ngForm).

Add similar `[(ngModel)]` bindings and `name` attributes to *Alter Ego* and *Hero Power*. You'll ditch the input box binding message and add a new binding (at the top) to the component's `diagnostic` property.

Then you can confirm that two-way data binding works *for the entire hero model*.

After revision, the core of the form should look like this:

* Each input element has an `id` property that is used by the `label` element's `for` attribute to match the label to its input control. * Each input element has a `name` property that is required by Angular forms to register the control with the form.

If you run the app now and change every hero model property, the form might display like this:

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

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

Track control state and validity with *ngModel*

Using `ngModel` in a form gives you more than just two-way data binding. It also tells you if the user touched the control, if the value changed, or if the value became invalid.

The *NgModel* directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state. You can leverage those class names to change the appearance of the control.

State	Class if true	Class if false
The control has been visited.	`ng-touched`	`ng-untouched`
The control's value has changed.	`ng-dirty`	`ng-pristine`
The control's value is valid.	`ng-valid`	`ng-invalid`

Temporarily add a template reference variable named `spy` to the *Name* `<input>` tag and use it to display the input's CSS classes.

Now run the app and look at the *Name* input box. Follow these steps *precisely*:

1. Look but don't touch.
2. Click inside the name box, then click outside it.
3. Add slashes to the end of the name.
4. Erase the name.

The actions and effects are as follows:

You should see the following transitions and class names:

The `ng-valid` / `ng-invalid` pair is the most interesting, because you want to send a strong visual signal when the values are invalid. You also want to mark required fields. To create such visual feedback, add definitions for the `ng-*` CSS classes.

Delete the `#spy` template reference variable and the `TODO` as they have served their purpose.

Add custom CSS for visual feedback

You can mark required fields and invalid data at the same time with a colored bar on the left of the input box:

You achieve this effect by adding these class definitions to a new `forms.css` file that you add to the project as a sibling to `index.html` :

Update the `<head>` of `index.html` to include this style sheet:

Show and hide validation error messages

You can improve the form. The *Name* input box is required and clearing it turns the bar red. That says something is wrong but the user doesn't know *what* is wrong or what to do about it. Leverage the control's state to reveal a helpful message.

When the user deletes the name, the form should look like this:

To achieve this effect, extend the `<input>` tag with the following:

- A template reference variable.
- The "*is required*" message in a nearby `<div>` , which you'll display only if the control is invalid.

Here's an example of an error message added to the *name* input box:

You need a template reference variable to access the input box's Angular control from within the template. Here you created a variable called `name` and gave it the value "ngModel".

Why "ngModel"? A directive's [exportAs](api/core/Directive) property tells Angular how to link the reference variable to the directive. You set `name` to `ngModel` because the `ngModel` directive's `exportAs` property happens to be "ngModel".

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

In this example, you 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 developers want the message to display at all times. If you ignore the `pristine` state, you would hide the message only when the value is valid. If you arrive in this component with a new (blank) hero or an invalid hero, you'll see the error message immediately, before you've done anything.

Some developers want the message to display only when the user makes an invalid change. Hiding the message while the control is "pristine" achieves that goal. You'll see the significance of this choice when you add a new hero to the form.

The hero *Alter Ego* is optional so you can leave that be.

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

Now you'll add a new hero in this form. Place a *New Hero* button at the bottom of the form and bind its click event to a `newHero` component method.

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; you haven't changed anything yet.

Enter a name and click *New Hero* again. The app displays a *Name is required* error message. You don't want error messages when you create a new (empty) hero. Why are you getting one now?

Inspecting the element in the browser tools reveals that the *name* input box is *no longer pristine*. The form remembers that you entered a name before clicking *New Hero*. Replacing the hero object *did not restore the pristine state* of the form controls.

You have to clear all of the flags imperatively, which you can do by calling the form's `reset()` method after calling the `newHero()` method.

Now clicking "New Hero" resets both the form and its control flags.

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, bind the form's `ngSubmit` event property to the hero form component's `onSubmit()` method:

You'd already defined a template reference variable, `#heroForm` , and initialized it with the value "ngForm". Now, use that variable to access the form with the Submit button.

You'll bind the form's overall validity via the `heroForm` variable to the button's `disabled` property using an event binding. Here's the code:

If you run the application now, you find that the button is enabled—although it doesn't do anything useful yet.

Now if you delete the Name, you violate the "required" rule, which is duly noted in the error message. The *Submit* button is also disabled.

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

For you, it was as simple as this:

1. Define a template reference variable on the (enhanced) form element.
2. Refer to that variable in a button many 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 you anything new about forms. But this is an opportunity to exercise some of your newly won binding skills. If you aren't interested, skip to this page's conclusion.

For a more strikingly visual effect, hide the data entry area and display something else.

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

The main form is visible from the start because the `submitted` property is false until you submit the form, as this fragment from the `HeroFormComponent` shows:

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

Now the app needs to show something else while the form is in the submitted state. Add the following HTML below the `<div>` wrapper you just wrote:

There's the hero again, displayed read-only with interpolation bindings. This `<div>` appears only while the component is in the submitted state.

The HTML includes an *Edit* button whose click event is bound to an expression that clears the `submitted` flag.

When you click the *Edit* button, this block disappears and the editable form reappears.

Summary

The Angular form discussed in this page 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.
- Handling form submission by binding to the `NgForm.ngSubmit` event property.
- Template-reference variables such as `#heroForm` and `#name`.
- `[(ngModel)]` syntax for two-way data binding.
- The use of `name` attributes for validation and form-element change tracking.
- The reference 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.

Here's the code for the final version of the application: