Building a template-driven form

{@a template-driven}

This tutorial shows you how to create a template-driven form whose control elements are bound to data properties, with input validation to maintain data integrity and styling to improve the user experience.

Template-driven forms use two-way data binding to update the data model in the component as changes are made in the template and vice versa.

Angular supports two design approaches for interactive forms. You can build forms by writing templates using Angular template syntax and directives with the form-specific directives and techniques described in this tutorial, or you can use a reactive (or model-driven) approach to build forms.

Template-driven forms are suitable for small or simple forms, while reactive forms are more scalable and suitable for complex forms. For a comparison of the two approaches, see Introduction to Forms

You can build almost any kind of form with an Angular template—login forms, contact forms, and pretty much any business form. You can lay out the controls creatively and bind them to the data in your object model. You can specify validation rules and display validation errors, conditionally enable or disable specific controls, trigger built-in visual feedback, and much more.

This tutorial shows you how to build a form from scratch, using a simplified sample form like the one from the Tour of Heroes tutorial to illustrate the techniques.

Run or download the example app: .

Objectives

This tutorial teaches you how to do the following:

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

Prerequisites

Before going further into template-driven forms, you should have a basic understanding of the following.

- TypeScript and HTML5 programming.
- Angular app-design fundamentals, as described in Angular Concepts.
- The basics of Angular template syntax.
- The form-design concepts that are presented in Introduction to Forms.

{@a intro}

Build a template-driven form

Template-driven forms rely on directives defined in the `FormsModule`.

- The `NgModel` directive reconciles value changes in the attached form element with changes in the data model, allowing you to respond to user input with input validation and error handling.

- The `NgForm` directive creates a top-level `FormGroup` instance and binds it to a `<form>` element to track aggregated form value and validation status. As soon as you import `FormsModule`, this directive becomes active by default on all `<form>` tags. You don't need to add a special selector.
- The `NgModelGroup` directive creates and binds a `FormGroup` instance to a DOM element.

The sample application

The sample form in this guide is used by the *Hero Employment Agency* to maintain personal information about heroes. Every hero needs a job. This form helps the agency match the right hero with the right crisis.

Clean Form

The form highlights some design features that make it easier to use. For instance, the two required fields have a green bar on the left to make them easy to spot. These fields have initial values, so the form is valid and the **Submit** button is enabled.

As you work with this form, you will learn how to include validation logic, how to customize the presentation with standard CSS, and how to handle error conditions to ensure valid input. If the user deletes the hero name, for example, the form becomes invalid. The application detects the changed status, and displays a validation error in an attention-grabbing style. In addition, the **Submit** button is disabled, and the "required" bar to the left of the input control changes from green to red.

Invalid, Name Required

Step overview

In the course of this tutorial, you bind a sample form to data and handle user input using the following steps.

1. Build the basic form.
   - Define a sample data model.
   - Include required infrastructure such as the `FormsModule`.
2. Bind form controls to data properties using the `ngModel` directive and two-way data-binding syntax.
   - Examine how `ngModel` reports control states using CSS classes.
   - Name controls to make them accessible to `ngModel`.
3. Track input validity and control status using `ngModel`.
   - Add custom CSS to provide visual feedback on the status.
   - Show and hide validation-error messages.
4. Respond to a native HTML button-click event by adding to the model data.
5. Handle form submission using the `ngSubmit` output property of the form.
   - Disable the **Submit** button until the form is valid.
   - After submit, swap out the finished form for different content on the page.

{@a step1}

Build the form

You can recreate the sample application from the code provided here, or you can examine or download the .

1. The provided sample application creates the `Hero` class which defines the data model reflected in the form.

2. The form layout and details are defined in the `HeroFormComponent` class.

   The component's `selector` value of "app-hero-form" means you can drop this form in a parent

template using the `<app-hero-form>` tag.

3. The following code creates a new hero instance, so that the initial form can show an example hero.

   This demo uses dummy data for `model` and `powers`. In a real app, you would inject a data service to get and save real data, or expose these properties as inputs and outputs.

4. The application enables the Forms feature and registers the created form component.

5. The form is displayed in the application layout defined by the root component's template.

   The initial template defines the layout for a form with two form groups and a submit button. The form groups correspond to two properties of the Hero data model, name and alterEgo. Each group has a label and a box for user input.

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

   The **Submit** button has some classes on it for styling. At this point, the form layout is all plain HTML5, with no bindings or directives.

6. The sample form uses some style classes from Twitter Bootstrap: `container`, `form-group`, `form-control`, and `btn`. To use these styles, the application's style sheet imports the library.

7. The form makes the hero applicant choose one superpower from a fixed list of agency-approved powers. The predefined list of `powers` is part of the data model, maintained internally in `HeroFormComponent`. The Angular NgForOf directive iterates over the data values to populate the `<select>` element.

If you run the application right now, you see the list of powers in the selection control. The input elements are not yet bound to data values or events, so they are still blank and have no behavior.

Early form with no binding

{@a ngModel}

Bind input controls to data properties

The next step is to bind the input controls to the corresponding `Hero` properties with two-way data binding, so that they respond to user input by updating the data model, and also respond to programmatic changes in the data by updating the display.

The `ngModel` directive declared in the `FormsModule` lets you bind controls in your template-driven form to properties in your data model. When you include the directive using the syntax for two-way data binding, `[(ngModel)]`, Angular can track the value and user interaction of the control and keep the view synced with the model.

1. Edit the template file `hero-form.component.html`.

2. Find the `<input>` tag next to the **Name** label.

3. Add the `ngModel` directive, using two-way data binding syntax `[(ngModel)]="..."`.

This example has a temporary diagnostic interpolation after each input tag, `{{model.name}}`, to show the current data value of the corresponding property. The note reminds you to remove the diagnostic lines when you have finished observing the two-way data binding at work.

{@a ngForm}

Access the overall form status

When you imported the `FormsModule` in your component, Angular automatically created and attached an NgForm directive to the `<form>` tag in the template (because `NgForm` has the selector `form` that matches `<form>` elements).

To get access to the `NgForm` and the overall form status, declare a template reference variable.

1. Edit the template file `hero-form.component.html`.

2. Update the `<form>` tag with a template reference variable, `#heroForm`, and set its value as follows.

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

3. Run the app.

4. Start typing in the **Name** input box.

As you add and delete characters, you can see them appear and disappear from the data model. For example:

ngModel in action

The diagnostic line that shows interpolated values demonstrates that values are really flowing from the input box to the model and back again.

Naming control elements

When you use `[(ngModel)]` on an element, you must define a `name` attribute for that element. Angular uses the assigned name to register the element with the `NgForm` directive attached to the parent `<form>` element.

The example added a `name` attribute to the `<input>` element 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.

1. Add similar `[(ngModel)]` bindings and `name` attributes to **Alter Ego** and **Hero Power**.

2. You can now remove the diagnostic messages that show interpolated values.

3. To confirm that two-way data binding works for the entire hero model, add a new text binding with the `json` pipe (which would serialize the data to a string) at the top to the component's template.

After these revisions, the form template should look like the following:

- Notice that each `<input>` element has an `id` property. This is used by the `<label>` element's `for` attribute to match the label to its input control. This is a standard HTML feature.

- Each `<input>` element also has the required `name` property that Angular uses to register the control with the form.

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

ngModel in action

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

4. When you have observed the effects, you can delete the `{{ model | json }}` text binding.

Track control states

The `NgModel` directive on a control tracks the state of that control. It tells you if the user touched the control, if the value changed, or if the value became invalid. Angular sets special CSS classes on the control element to reflect the state, as shown in the following table.

```
<th>
  State
</th>

<th>
  Class if true
</th>

<th>
  Class if false
</th>
```

```
<td>
  The control has been visited.
</td>

<td>
  <code>ng-touched</code>
</td>

<td>
  <code>ng-untouched</code>
</td>
```

```
<td>
  The control's value has changed.
</td>

<td>
  <code>ng-dirty</code>
</td>

<td>
```

```
  <code>ng-pristine</code>
</td>
```

```
<td>
  The control's value is valid.
</td>

<td>
  <code>ng-valid</code>
</td>

<td>
  <code>ng-invalid</code>
</td>
```

Additionally, Angular applies the `ng-submitted` class to `<form>` elements upon submission. This class does *not* apply to inner controls.

You use these CSS classes to define the styles for your control based on its status.

Observe control states

To see how the classes are added and removed by the framework, open the browser's developer tools and inspect the `<input>` element that represents the hero name.

1. Using your browser's developer tools, find the `<input>` element that corresponds to the **Name** input box. You can see that the element has multiple CSS classes in addition to "form-control".

2. When you first bring it up, the classes indicate that it has a valid value, that the value has not been changed since initialization or reset, and that the control has not been visited since initialization or reset.

   ```
   <input ... class="form-control ng-untouched ng-pristine ng-valid" ...>
   ```

3. Take the following actions on the **Name** `<input>` box, and observe which classes appear.

   - Look but don't touch. The classes indicate that it is untouched, pristine, and valid.
   - Click inside the name box, then click outside it. The control has now been visited, and the element has the `ng-touched` class instead of the `ng-untouched` class.
   - Add slashes to the end of the name. It is now touched and dirty.
   - Erase the name. This makes the value invalid, so the `ng-invalid` class replaces the `ng-valid` class.

Create visual feedback for states

The `ng-valid` / `ng-invalid` pair is particularly interesting, because you want to send a strong visual signal when the values are invalid. You also want to mark required fields.

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

Invalid Form

To change the appearance in this way, take the following steps.

1. Add definitions for the `ng-*` CSS classes.

2. Add these class definitions to a new `forms.css` file.

3. Add the new file to the project as a sibling to `index.html` :

4. In the `index.html` file, update the `<head>` tag to include the new style sheet.

Show and hide validation error messages

The **Name** input box is required and clearing it turns the bar red. That indicates that something is wrong, but the user doesn't know what is wrong or what to do about it. You can provide a helpful message by checking for and responding to the control's state.

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

Name required

The **Hero Power** select box is also required, but it doesn't need this kind of error handling because the selection box already constrains the selection to valid values.

To define and show an error message when appropriate, take the following steps.

1. Extend the `<input>` tag with a template reference variable that you can use to access the input box's Angular control from within the template. In the example, the variable is `#name="ngModel"` .

   The template reference variable (`#name`) is set to `"ngModel"` because that is the value of the [`NgModel.exportAs`] property. This property tells Angular how to link a reference variable to a directive.

2. Add a `<div>` that contains a suitable error message.

3. Show or hide the error message by binding properties of the `name` control to the message `<div>` element's `hidden` property.

4. Add a conditional error message to the *name* input box, as in the following example.

Illustrating the "pristine" state

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

You might want the message to display only when the user makes an invalid change. Hiding the message while the control is in the `pristine` state achieves that goal. You'll see the significance of this choice when you add a new hero to the form in the next step.

Add a new hero

This exercise shows how you can respond to a native HTML button-click event by adding to the model data. To let form users add a new hero, you will add a **New Hero** button that responds to a click event.

1. In the template, place a "New Hero" `<button>` element at the bottom of the form.

2. In the component file, add the hero-creation method to the hero data model.

3. Bind the button's click event to a hero-creation method, `newHero()` .

4. Run the application again and click the **New Hero** button.

   The form clears, and the *required* bars to the left of the input box are red, indicating invalid `name` and `power` properties. Notice that the error messages are hidden. This is because the form is pristine; you haven't changed anything yet.

5. Enter a name and click **New Hero** again.

   Now the application displays a *Name is required* error message, because the input box is no longer pristine. The form remembers that you entered a name before clicking **New Hero**.

6. To restore the pristine state of the form controls, clear all of the flags imperatively by calling the form's `reset()` method after calling the `newHero()` method.

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

See the User Input guide for more information about listening for DOM events with an event binding and updating a corresponding component property.

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 does trigger a form-submit event because of its type (`type="submit"`). To respond to this event, take the following steps.

1. Bind the form's `ngSubmit` event property to the hero-form component's `onSubmit()` method.

2. Use the template reference variable, `#heroForm` to access the form that contains the **Submit** button and create an event binding. You will bind the form property that indicates its overall validity to the **Submit** button's `disabled` property.

3. Run the application now. Notice that the button is enabled—although it doesn't do anything useful yet.

4. Delete the **Name** value. This violates the "required" rule, so it displays the error message—and notice that it also disables the **Submit** button.

   You didn't have to explicitly wire the button's enabled state to the form's validity. The `FormsModule` did this automatically when you defined a template reference variable on the enhanced form element, then referred to that variable in the button control.

Respond to form submission

To show a response to form submission, you can hide the data entry area and display something else in its place.

1. Wrap the entire 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.

2. To show something else while the form is in the submitted state, add the following HTML below the new `<div>` wrapper.

   This `<div>`, which shows a read-only hero with interpolation bindings, appears only while the component is in the submitted state.

   The alternative display includes an *Edit* button whose click event is bound to an expression

that clears the `submitted` flag.

3. Click the *Edit* button to switch the display back to the editable form.

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 or 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: