Building dynamic forms

Contents ›

Many forms, such as questionaires, can be very similar to one another in format and intent. To make it faster and easier to generate different versions of such a form, you can create a *dynamic form template* based on metadata that describes the business object model. You can then use the template to generate new forms automatically, according to changes in the data model.

The technique is particularly useful when you have a type of form whose content must change frequently to meet rapidly changing business and regulatory requirements. A typical use case is a questionaire. You might need to get input from users in different contexts. The format and style of the forms a user sees should remain constant, while the actual questions you need to ask vary with the context.

In this tutorial you will build a dynamic form that presents a basic questionaire. You will build an online application for heroes seeking employment. The agency is constantly tinkering with the application process, but by using the dynamic form you can create the new forms on the fly without changing the application code.

The tutorial walks you through the following steps.

1. Enable reactive forms for a project.
2. Establish a data model to represent form controls.
3. Populate the model with sample data.
4. Develop a component to create form controls dynamically.

The form you create uses input validation and styling to improve the user experience. It has a Submit button that is only enabled when all user input is valid, and flags invalid input with color coding and error messages.

The basic version can evolve to support a richer variety of questions, more graceful rendering, and superior user experience.

> See the live example / download example.

Prerequisites

Before doing this tutorial, you should have a basic understanding to the following.

- TypeScript and HTML5 programming.
- Fundamental concepts of Angular app design.
- Basic knowledge of reactive forms.

Enable reactive forms for your project

Dynamic forms are based on reactive forms. To give the application access reactive forms directives, the root module imports `ReactiveFormsModule` from the `@angular/forms` library.

The following code from the example shows the setup in the root module.

app.module.ts | main.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { DynamicFormComponent } from './dynamic-form.component';
import { DynamicFormQuestionComponent } from './dynamic-form-question.component';

@NgModule({
  imports: [ BrowserModule, ReactiveFormsModule ],
  declarations: [ AppComponent, DynamicFormComponent, DynamicFormQuestionComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule {
  constructor() {
  }
}
```

Create a form object model

A dynamic form requires an object model that can describe all scenarios needed by the form functionality. The example hero-application form is a set of questions—that is, each control in the form must ask a question and accept an answer.

The data model for this type of form must represent a question. The example includes the `DynamicFormQuestionComponent`, which defines a question as the fundamental object in the model.

The following `QuestionBase` is a base class for a set of controls that can represent the question and its answer in the form.

src/app/question-base.ts

```
export class QuestionBase<T> {
```

```
  value: T;
  key: string;
  label: string;
  required: boolean;
  order: number;
  controlType: string;
  type: string;
  options: {key: string, value: string}[];

  constructor(options: {
      value?: T;
      key?: string;
      label?: string;
      required?: boolean;
      order?: number;
      controlType?: string;
      type?: string;
      options?: {key: string, value: string}[];
    } = {}) {
    this.value = options.value;
    this.key = options.key || '';
    this.label = options.label || '';
    this.required = !!options.required;
    this.order = options.order === undefined ? 1 : options.order;
    this.controlType = options.controlType || '';
    this.type = options.type || '';
    this.options = options.options || [];
  }
}
```

Define control classes

From this base, the example derives two new classes, `TextboxQuestion` and `DropdownQuestion`, that represent different control types. When you create the form template in the next step, you will instantiate these specific question types in order to render the appropriate controls dynamically.

- The `TextboxQuestion` control type presents a question and allows users to enter input.

  src/app/question-textbox.ts

```
  import { QuestionBase } from './question-base';

  export class TextboxQuestion extends QuestionBase<string> {
    controlType = 'textbox';
  }
```

The `TextboxQuestion` control type will be represented in a form template using an `<input>` element. The `type` attribute of the element will be defined based on the `type` field specified in the `options` argument (for example `text`, `email`, `url`).

- The `DropdownQuestion` control presents a list of choices in a select box.

src/app/question-dropdown.ts

```
import { QuestionBase } from './question-base';

export class DropdownQuestion extends QuestionBase<string> {
  controlType = 'dropdown';
}
```

Compose form groups

A dynamic form uses a service to create grouped sets of input controls, based on the form model. The following `QuestionControlService` collects a set of `FormGroup` instances that consume the metadata from the question model. You can specify default values and validation rules.

src/app/question-control.service.ts

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

import { QuestionBase } from './question-base';

@Injectable()
export class QuestionControlService {
  constructor() { }

  toFormGroup(questions: QuestionBase<string>[] ) {
    const group: any = {};

    questions.forEach(question => {
      group[question.key] = question.required ? new FormControl(question.value || '',
Validators.required)
                                              : new FormControl(question.value || '');
    });
    return new FormGroup(group);
  }
}
```

Compose dynamic form contents

The dynamic form itself will be represented by a container component, which you will add in a later step. Each question is represented in the form component's template by an `<app-question>` tag, which matches an instance of `DynamicFormQuestionComponent`.

The `DynamicFormQuestionComponent` is responsible for rendering the details of an individual question based on values in the data-bound question object. The form relies on a `[formGroup]` directive to connect the template HTML to the underlying control objects. The `DynamicFormQuestionComponent` creates form groups and populates them with controls defined in the question model, specifying display and validation rules.

dynamic-form-question.component.html | dynamic-form-question.component.ts

```
<div [formGroup]="form">
  <label [attr.for]="question.key">{{question.label}}</label>

  <div [ngSwitch]="question.controlType">

    <input *ngSwitchCase="'textbox'" [formControlName]="question.key"
           [id]="question.key" [type]="question.type">

    <select [id]="question.key" *ngSwitchCase="'dropdown'"
[formControlName]="question.key">
      <option *ngFor="let opt of question.options" [value]="opt.key">{{opt.value}}
</option>
    </select>

  </div>

  <div class="errorMessage" *ngIf="!isValid">{{question.label}} is required</div>
</div>
```

The goal of the `DynamicFormQuestionComponent` is to present question types defined in your model. You only have two types of questions at this point but you can imagine many more. The `ngSwitch` statement in the template determines which type of question to display. The switch uses directives with the `formControlName` and `formGroup` selectors. Both directives are defined in `ReactiveFormsModule`.

Supply data

Another service is needed to supply a specific set of questions from which to build an individual form. For this exercise you will create the `QuestionService` to supply this array of questions from the hard-coded sample data. In a real-world app, the service might fetch data from a backend system. The key point, however, is that you control the hero job-application questions entirely through the objects returned from `QuestionService`. To maintain the questionnaire as requirements change, you only need to add, update, and remove objects from the `questions` array.

The `QuestionService` supplies a set of questions in the form of an array bound to `@Input()` questions.

src/app/question.service.ts

```ts
import { Injectable } from '@angular/core';

import { DropdownQuestion } from './question-dropdown';
import { QuestionBase } from './question-base';
import { TextboxQuestion } from './question-textbox';
import { of } from 'rxjs';

@Injectable()
export class QuestionService {

  // TODO: get from a remote source of question metadata
  getQuestions() {

    const questions: QuestionBase<string>[] = [

      new DropdownQuestion({
        key: 'brave',
        label: 'Bravery Rating',
        options: [
          {key: 'solid',  value: 'Solid'},
          {key: 'great',  value: 'Great'},
          {key: 'good',   value: 'Good'},
          {key: 'unproven', value: 'Unproven'}
        ],
        order: 3
      }),

      new TextboxQuestion({
        key: 'firstName',
        label: 'First name',
        value: 'Bombasto',
        required: true,
        order: 1
      }),

      new TextboxQuestion({
        key: 'emailAddress',
        label: 'Email',
        type: 'email',
        order: 2
      })
    ];

    return of(questions.sort((a, b) => a.order - b.order));
  }
}
```

Create a dynamic form template

The `DynamicFormComponent` component is the entry point and the main container for the form, which is represented using the `<app-dynamic-form>` in a template.

The `DynamicFormComponent` component presents a list of questions by binding each one to an `<app-question>` element that matches the `DynamicFormQuestionComponent`.

dynamic-form.component.html | dynamic-form.component.ts

```
<div>
  <form (ngSubmit)="onSubmit()" [formGroup]="form">

    <div *ngFor="let question of questions" class="form-row">
      <app-question [question]="question" [form]="form"></app-question>
    </div>

    <div class="form-row">
      <button type="submit" [disabled]="!form.valid">Save</button>
    </div>
  </form>

  <div *ngIf="payLoad" class="form-row">
    <strong>Saved the following values</strong><br>{{payLoad}}
  </div>
</div>
```

Display the form

To display an instance of the dynamic form, the `AppComponent` shell template passes the `questions` array returned by the `QuestionService` to the form container component, `<app-dynamic-form>`.

app.component.ts

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

import { QuestionService } from './question.service';
import { QuestionBase } from './question-base';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
    <div>
      <h2>Job Application for Heroes</h2>
      <app-dynamic-form [questions]="questions$ | async"></app-dynamic-form>
    </div>
```

```
  `,
  providers:  [QuestionService]
})
export class AppComponent {
  questions$: Observable<QuestionBase<any>[]>;

  constructor(service: QuestionService) {
    this.questions$ = service.getQuestions();
  }
}
```

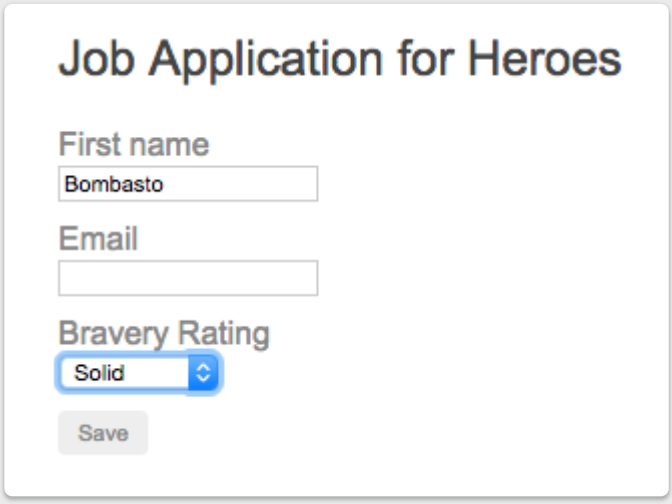
The example provides a model for a job application for heroes, but there are no references to any specific hero question other than the objects returned by `QuestionService`. This separation of model and data allows you to repurpose the components for any type of survey as long as it's compatible with the *question* object model.

Ensuring valid data

The form template uses dynamic data binding of metadata to render the form without making any hardcoded assumptions about specific questions. It adds both control metadata and validation criteria dynamically.

To ensure valid input, the *Save* button is disabled until the form is in a valid state. When the form is valid, you can click *Save* and the app renders the current form values as JSON.

The following figure shows the final form.



Next steps

- **Different types of forms and control collection**
  This tutorial shows how to build a a questionaire, which is just one kind of dynamic form. The example uses `FormGroup` to collect a set of controls. For an example of a different type of dynamic form, see the section Creating dynamic forms in the Reactive Forms guide. That example also shows how to use `FormArray` instead of `FormGroup` to collect a set of controls.
- **Validating user input**
  The section Validating form input introduces the basics of how input validation works in reactive forms.

The Form validation guide covers the topic in more depth.

Version 10.0.10-local+sha.b32126c335.