



Dynamic forms

Building handcrafted forms can be costly and time-consuming, especially if you need a great number of them, they're similar to each other, and they change frequently to meet rapidly changing business and regulatory requirements.

It may be more economical to create the forms dynamically, based on metadata that describes the business object model.

This cookbook shows you how to use `FormGroup` to dynamically render a simple form with different control types and validation. It's a primitive start. It might evolve to support a much richer variety of questions, more graceful rendering, and superior user experience. All such greatness has humble beginnings.

The example in this cookbook is a dynamic form to build an online application experience for heroes seeking employment. The agency is constantly tinkering with the application process. You can create the forms on the fly *without changing the application code*.

See the [live example](#) / [download example](#).

Bootstrap

Start by creating an `NgModule` called `AppModule`.

This cookbook uses [reactive forms](#).

Reactive forms belongs to a different `NgModule` called `ReactiveFormsModule`, so in order to access any reactive forms directives, you have to import `ReactiveFormsModule` from the `@angular/forms` library.

Bootstrap the `AppModule` in `main.ts`.

`app.module.ts`

`main.ts`

```
import { BrowserModule }
import { ReactiveFormsModule }
import { NgModule }
```

```
import { AppComponent }
```

```
from '@angular/platform-browser';
from '@angular/forms';
from '@angular/core';
```

```
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() {
  }
}
```

Question model

The next step is to define an object model that can describe all scenarios needed by the form functionality. The hero application process involves a form with a lot of questions. The *question* is the most fundamental object in the model.

The following `QuestionBase` is a fundamental question class.

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
  } = {}) {
    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 || '';
  }
}
```

From this base you can derive two new classes in [TextboxQuestion](#) and [DropdownQuestion](#) that represent textbox and dropdown questions. The idea is that the form will be bound to specific question types and render the appropriate controls dynamically.

[TextboxQuestion](#) supports multiple HTML5 types such as text, email, and url via the [type](#) property.

src/app/question-textbox.ts

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

export class TextboxQuestion extends QuestionBase<string> {
  controlType = 'textbox';
  type: string;

  constructor(options: {} = {}) {
    super(options);
    this.type = options['type'] || '';
  }
}
```

[DropdownQuestion](#) 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';
  options: {key: string, value: string}[] = [];

  constructor(options: {} = {}) {
    super(options);
    this.options = options['options'] || [];
  }
}
```

Next is [QuestionControlService](#), a simple service for transforming the questions to a [FormGroup](#). In a nutshell, the form group consumes the metadata from the question model and allows you to 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>[] ) {
    let group: any = {};

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

Question form components

Now that you have defined the complete model you are ready to create components to represent the dynamic form.

`DynamicFormComponent` is the entry point and the main container for the form.

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>
  </form>
</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>

```

It presents a list of questions, each bound to a `<app-question>` component element. The `<app-question>` tag matches the `DynamicFormQuestionComponent`, the component responsible for rendering the details of each *individual* question based on values in the data-bound question object.

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>

```

Notice this component can present any type of question in your model. You only have two types of questions at this point but you can imagine many more. The `ngSwitch` determines which type of question to display.

In both components you're relying on Angular's **formGroup** to connect the template HTML to the underlying control objects, populated from the question model with display and validation rules.

`formControlName` and `formGroup` are directives defined in `ReactiveFormsModule`. The templates can access these directives directly since you imported `ReactiveFormsModule` from `AppModule`.

Questionnaire data

`DynamicFormComponent` expects the list of questions in the form of an array bound to `@Input() questions`.

The set of questions you've defined for the job application is returned from the `QuestionService`. In a real app you'd retrieve these questions from storage.

The key point is that you control the hero job application questions entirely through the objects returned from `QuestionService`. Questionnaire maintenance is a simple matter of adding, updating, and removing objects from the `questions` array.

src/app/question.service.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() {

        let 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));
  }
}

```

Finally, display an instance of the form in the `AppComponent` shell.

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();
  }
}

```

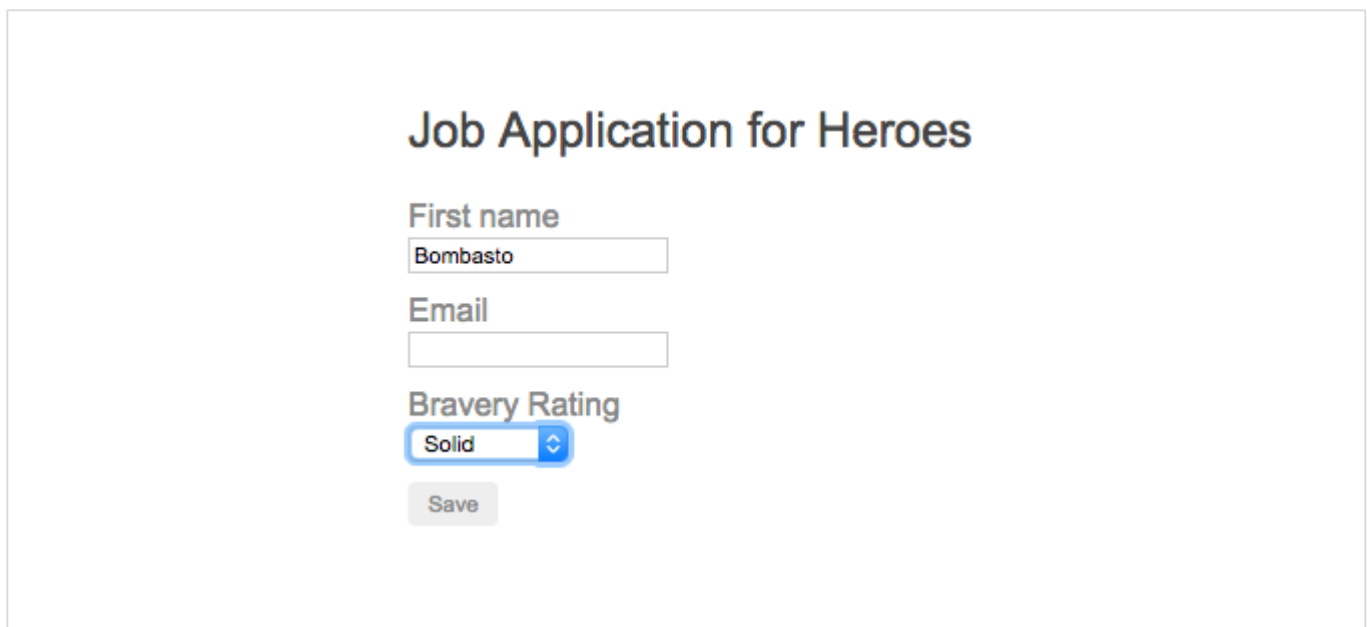
Dynamic Template

Although in this example you're modelling a job application for heroes, there are no references to any specific hero question outside the objects returned by `QuestionService`.

This is very important since it allows you to repurpose the components for any type of survey as long as it's compatible with the *question* object model. The key is the dynamic data binding of metadata used to render the form without making any hardcoded assumptions about specific questions. In addition to control metadata, you are also adding validation dynamically.

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. This proves that any user input is bound back to the data model. Saving and retrieving the data is an exercise for another time.

The final form looks like this:



Job Application for Heroes

First name

Email

Bravery Rating

[Back to top](#)