ANGULAR FORMS

VIVEKANAND P V

ANGULAR FORMS: CORE CONCEPTS

Two Variants:

1. Template-driven forms
2. Reactive forms

ANGULAR FORMS: CORE CONCEPTS

Prerequisites:

1. Import FormsModule for template forms
2. Import ReactiveFormsModule for reactive forms

ANGULAR FORMS: CORE CONCEPTS

Template-forms:

1. Trace their history to Angular.js
2. Nice fit for simple needs
3. Directives are everywhere: fields, validation, form!
4. Predominantly template-based

ANGULAR FORMS: CORE CONCEPTS

Reactive-forms:

1. Expose the observable-based API
2. Suitable for complex UI needs and validation
3. Predominantly code-based

ANGULAR FORMS: CORE CONCEPTS

Beginning

1. Start with a plain html form
2. Remove method and action, turn on novalidate, turn off autocomplete

ANGULAR FORMS: CORE CONCEPTS

Template forms: Step 1

1. Apply template variable to form and get ngForm exported to it #f="[ngForm]"
2. Listen to the built-in ngSubmit event (ngSubmit)="onFormPost(f.value)"

ANGULAR FORMS: CORE CONCEPTS

Template forms: Step 2

1. Apply ngModel directive to all input fields (name attribute is required)
2. For validation, every input field uses appropriate directives such as required, minlength, maxlength, etc

ANGULAR FORMS: CORE CONCEPTS

Template forms: Step 3

1. For validation error messages, export the input field as a template reference assigned with ngModel `#field="ngModel"`
2. Field, now has access to errors object. Use it to display proper error message

ANGULAR FORMS: CORE CONCEPTS

Every field thus exported with ngModel exposes prestine, touched, dirty, valid, and invalid Boolean flags, which we use to fine-tune the UX for validation errors

ANGULAR FORMS: CORE CONCEPTS

Field States

1. A field is prestine if user has not clicked or tabbed over
2. As soon as user clicks or tabs over or types, the field becomes touched
3. If the user types, the field becomes dirty
4. Valid and invalid depend on the validation directives applied to the field

ANGULAR FORMS: CORE CONCEPTS

Ideally, the template form should use the component for submit handler. We get the form value from the ngForm exported template-variable as `f.value`. Here ngSubmit is hooked to the component handler as

```
(ngSubmit)="onSubmitHandler(f.value)"
```

ANGULAR FORMS: CORE CONCEPTS

Typically, http access is provided by a service. This service is injected in the component and the form component invokes the proper form submission method in the service. Service in turn makes the proper Http call (post/put) to the backend.

ANGULAR FORMS: CORE CONCEPTS

Typical Use Cases for Template Forms

1. Login forms
2. Feedback forms
3. Dashboard drill-down forms
4. Report filtration forms

ANGULAR FORMS: CORE CONCEPTS

What it demands?

1. Working knowledge of directives

ANGULAR FORMS: CORE CONCEPTS

What is easy?

1. Predominantly template, so less code clutter in component
2. Easy to get started
3. Good for rapid prototyping of forms

ANGULAR FORMS: CORE CONCEPTS

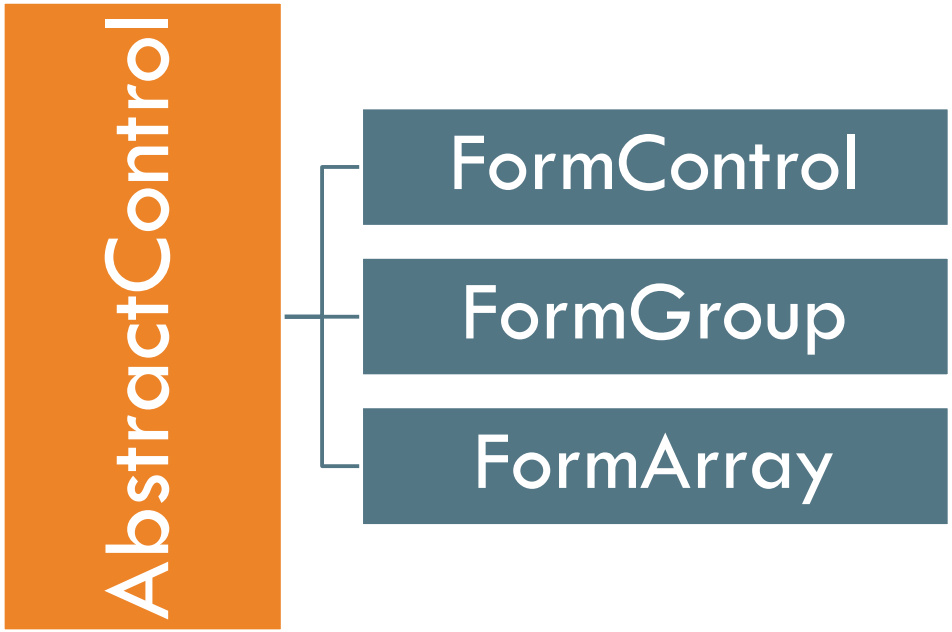
What is not easy?

1. Complex validations (multi-field, asynchronous)
2. Need to write custom error validator directives
3. Accomplishing complex UI capabilities such as field arrays

ANGULAR FORMS: CORE CONCEPTS

Reactive Forms

ANGULAR FORMS: CORE CONCEPTS



ANGULAR FORMS: CORE CONCEPTS

A reactive form starts journey in the component class as FormGroup. This can contain FormControls, FormArrays, or FormGroups. Extremely customizable.

ANGULAR FORMS: CORE CONCEPTS

A validator for a FormControl is applied at the time of instantiation (as function reference). A FormControl can have one or more (as array) validator functions. A FormControl can also have async validator/s as well.

ANGULAR FORMS: CORE CONCEPTS

Once the reactive form is created in the code, the html form in the template should be hooked using formGroup directive. ngSubmit should be used for form post handling. Since the form is created in the component, there is no form exported as a template variable.

ANGULAR FORMS: CORE CONCEPTS

Every input field is applied with formControlName directive and the name of the control has to be passed to it. This name should exactly match the one given at the time of FormGroup creation. Attribute ***name*** is not required.

ANGULAR FORMS: CORE CONCEPTS

For validation error drill-down, there are two approaches:

1. Use form.controls["control"].errors…
2. Expose the individual controls as getters and use them in the validation error blocks

ANGULAR FORMS: CORE CONCEPTS

1. Custom validators are easy as they are functions.
2. Any function that takes the AbstractControl object and returns validation error object can be a validator function.
3. Async validators return either a Promise or an Observable.
4. Beware of the `this` binding problem

ANGULAR FORMS: CORE CONCEPTS

1. A validator should return null in case of successful validation. Else, the { error: true }
2. Async validator, if uses Promise should never reject in any case. Use resolve in both cases. In validation success, resolve(null). Else resolve({ error: true })