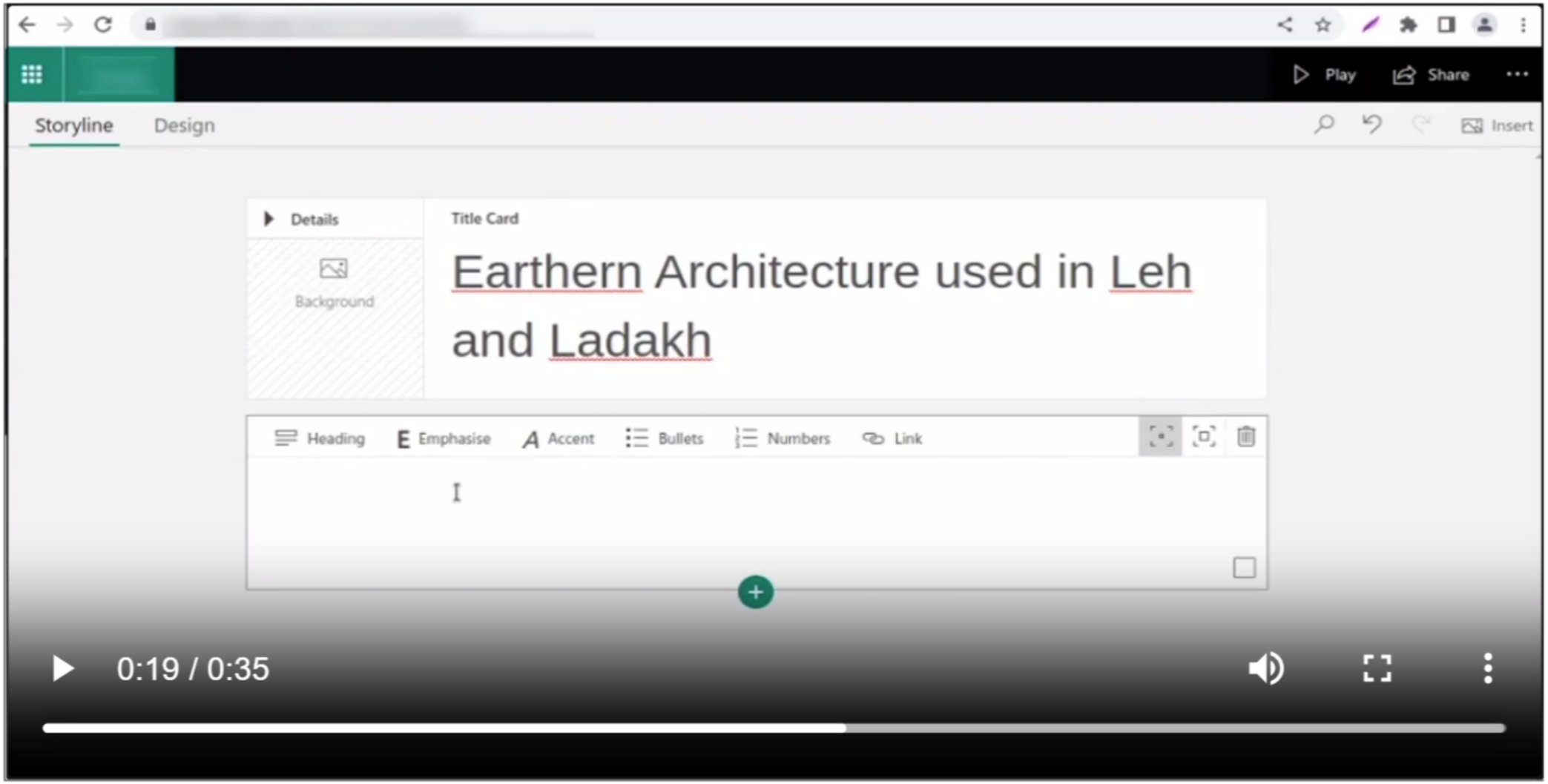
How is this form different from a normal form?

Slide Note

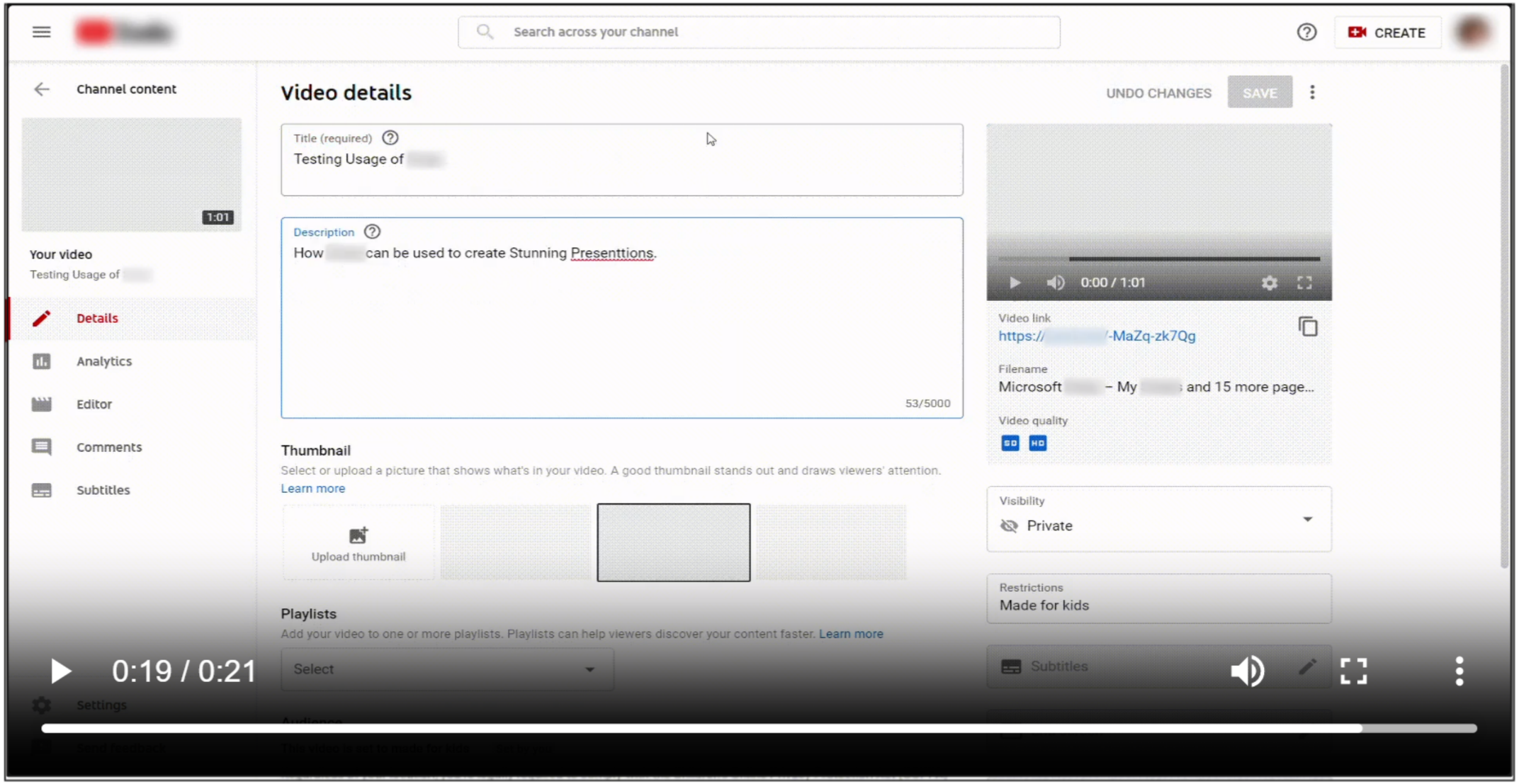
Notice that form values are added without a Submit button dynamically.



How are values being added in this form?

Slide Note

Let's observe another form where we edit some form values.



Slide Note

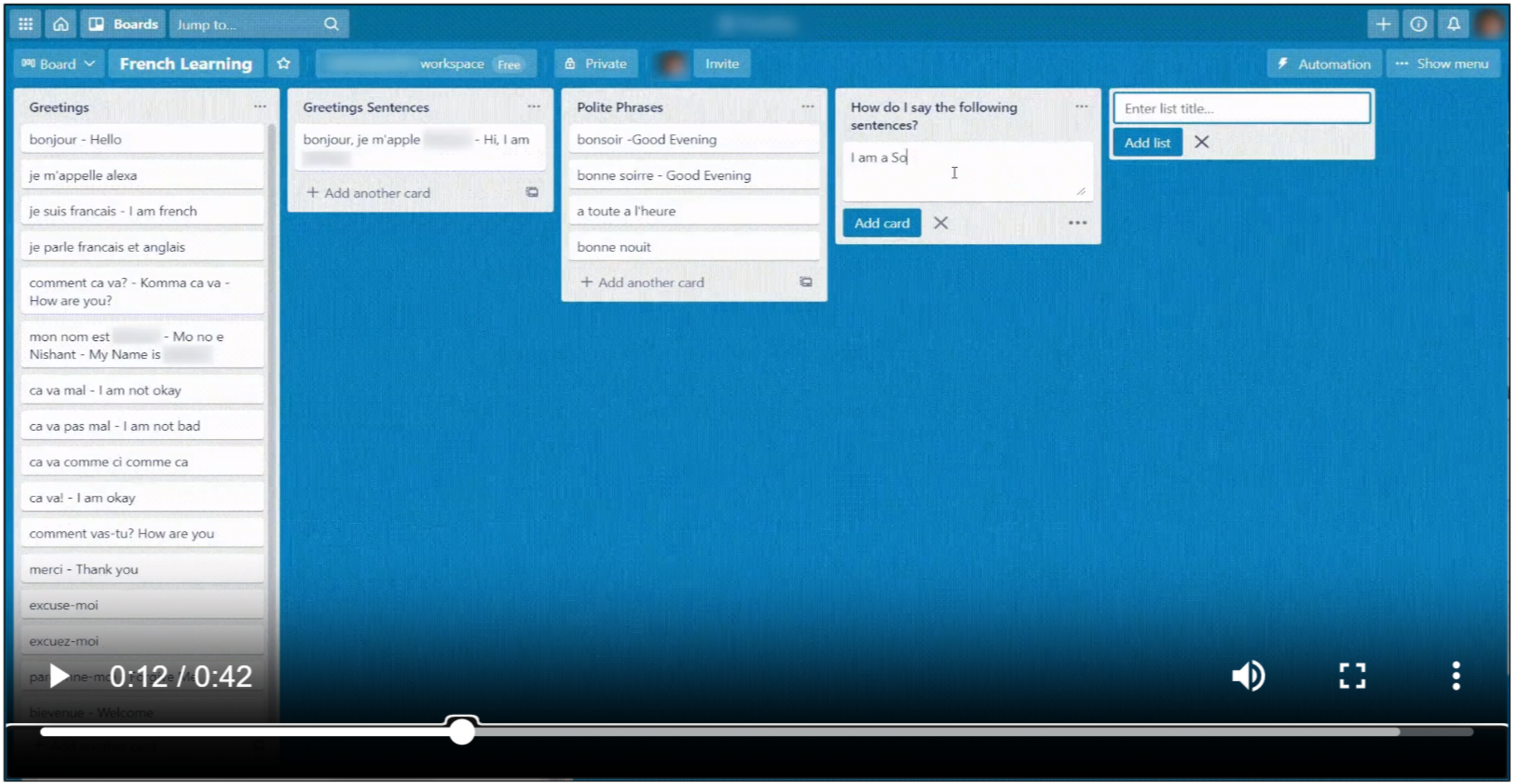
Did you notice in the form that values are saved automatically, and we can undo the changes to an earlier version?

Which two most important things did you notice while filling entries in the form fields?

Slide Note

Observe what is happening in this task list board when adding a new list.

Also observe while we are editing its content.



Slide Note

When contents have to be added in a form without transformation, there is no need for the Save button. Input is saved automatically.

□ But when some transformation in the content is required (here, markdown is getting transformed), a **Save** button is required.

□ A **Save** button transforms the hyphenedlist to bulleted-list when the content is edited.

What did you see on the trello board?

A list is getting dynamically added without the Save button.

The Save button transforms the hyphenated list into a bulleted list when editing.

Slide Note

Did you notice in the form that values are saved automatically, and we can undo the changes to an earlier version?

Angular provides reactive forms that update the form data when the underlying data changes.

Slide Note

Develop Interactive Reactive Forms Inside SPA

Slide Note

Learning Objectives

- Explain reactive forms in Angular
- Create, display, and replace `FormControl` values
- Analyze how data flows in reactive forms
- Contrast grouping form controls using `FormGroup` and `FormArray`
- Check the validity of fields in reactive forms using validator functions
- Add custom validators to reactive forms

Slide Note

Reactive (Model-Driven) Forms

- Reactive forms are one of the two ways to build forms in Angular.
- These are also called model-driven forms, where the structure of the form is defined in the component class and the HTML content changes depending on the code in the component.
- Reactive forms use an explicit and **immutable** approach to manage the state of a form at a given point in time.
- Each change to the form state returns a new state, which maintains the integrity of the model between changes.
- They are built around observable streams that provide form inputs and values as streams of input values that can be accessed **synchronously**.
- These forms thus provide:
  - synchronous access to the data model
  - immutability with observable operators
  - change tracking through observable streams

Slide Note

- In template-driven forms, keeping the template as the source of all the form validation rules is hard to read and maintain.
- In template-driven forms, as we add more and more validator tags to a field or when we start adding complex cross-field validations, the readability and maintainability of the form decrease.
- The Reactive approach removes the core validation logic from the template and hence makes the template code quite clean.
- Reactive forms keeps the data model pure by providing it as an immutable data structure, which is preferred by developers.
- Reactive transformations such as adding elements dynamically are possible in reactive forms.
- Since Reactive forms provide direct synchronous access to the data model, it makes creating large scale forms easier.
- From a unit testing perspective, it is easier to write unit tests with Reactive forms, since the logic is contained inside our component.

Benefits of Using Reactive Forms

Easy to maintain by keeping template code clean

- Removes core validation logic from the template

Create more scalable large forms easily

- Provides direct synchronous data access

Can add form fields dynamically

- Defines data model explicitly

Unit Testing without rendering the UI is possible

- Synchronous access to form and data models

Immutable form data structure

- Keeps data model pure, hence preferred by developers

Slide Note

• Reactive forms provide direct, explicit access to the underlying forms object model.

• Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable.

• If forms are a key part of our application, or we're already using reactive patterns for building our application, use reactive forms.

• Reactive forms are more scalable than template-driven forms.

• They provide direct access to the underlying form API, and use synchronous data flow between the view and the data model, which makes creating large-scale forms easier.

• Reactive forms require less setup for testing, and testing does not require deep understanding of change detection to properly test form updates and validation.

Reactive vs. Template-Driven Forms

	REACTIVE	TEMPLATE-DRIVEN
Setup of form model	explicit, created in component class	implicit, created by directives
Data model	structured and immutable	unstructured and mutable
Data flow	synchronous	asynchronous
Form validation	functions	directives
Scalability	Large forms can be created easily	Not reusable and works for simple scenarios only
Testing	Less setup for testing	More setup for testing as it requires detailed knowledge of change detection process

Slide Note

- Reactive forms provide direct, explicit access to the underlying forms object model.
- Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable.
- If forms are a key part of our application, or we're already using reactive patterns for building our application, use reactive forms.
- Reactive forms are more scalable than template-driven forms.
- They provide direct access to the underlying form API, and use synchronous data flow between the view and the data model, which makes creating large-scale forms easier.
- Reactive forms require less setup for testing, and testing does not require deep understanding of change detection to properly test form updates and validation.

Form Foundation Classes

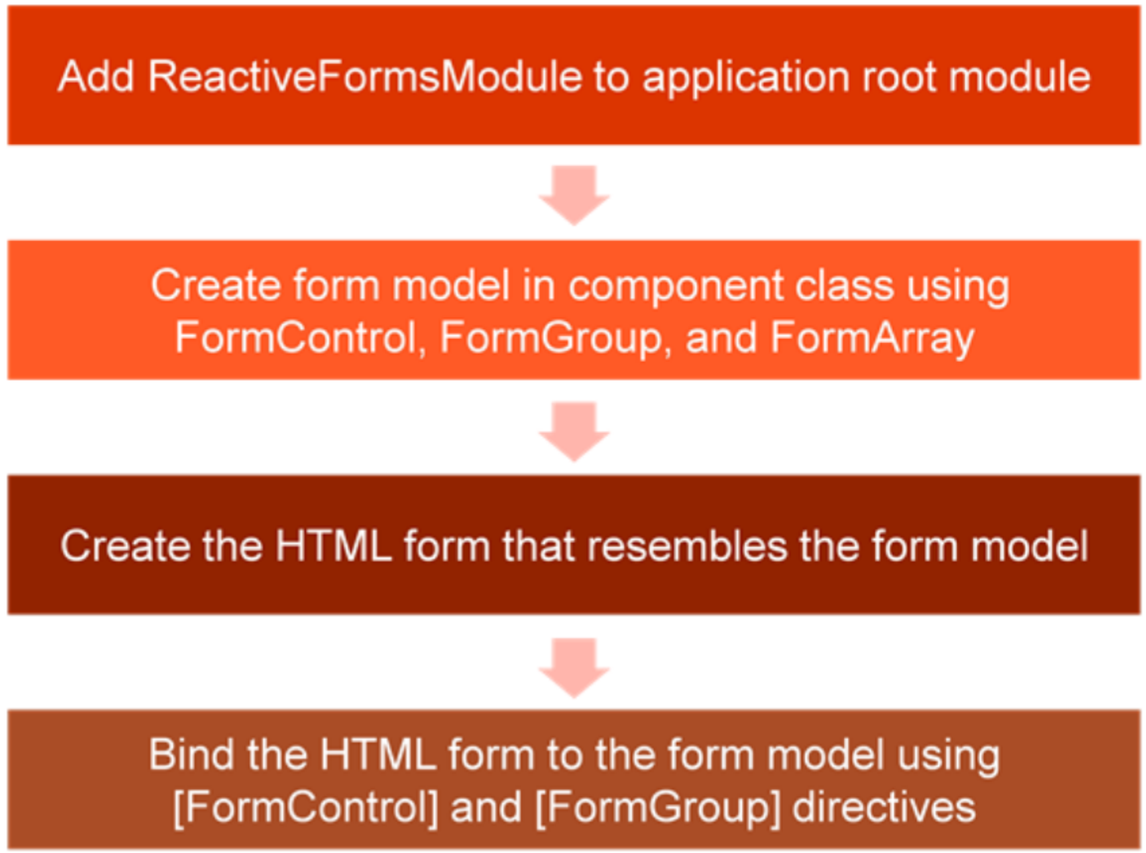
Reactive forms are built on the following base classes.

Base Classes	Details
FormControl	Tracks the value and validation status of an individual form control.
FormGroup	Tracks the value and validation status for a collection of form control instances.
FormArray	Tracks the value and validity state of an array of form control instances. Used for dynamically adding form control elements.
ControlValueAccessor	Creates a bridge between Angular FormControl instances and built-in DOM elements.

Slide Note

Steps For Creating Reactive Forms

Here's how to create reactive forms:



Slide Note

In reactive forms, the form model is the source of truth; it provides the value and status of the form element at any given point in time. This is done through the [formControl] directive on the input element.

Direct access to the form control instance is achieved after the link is created by FormControlDirective.

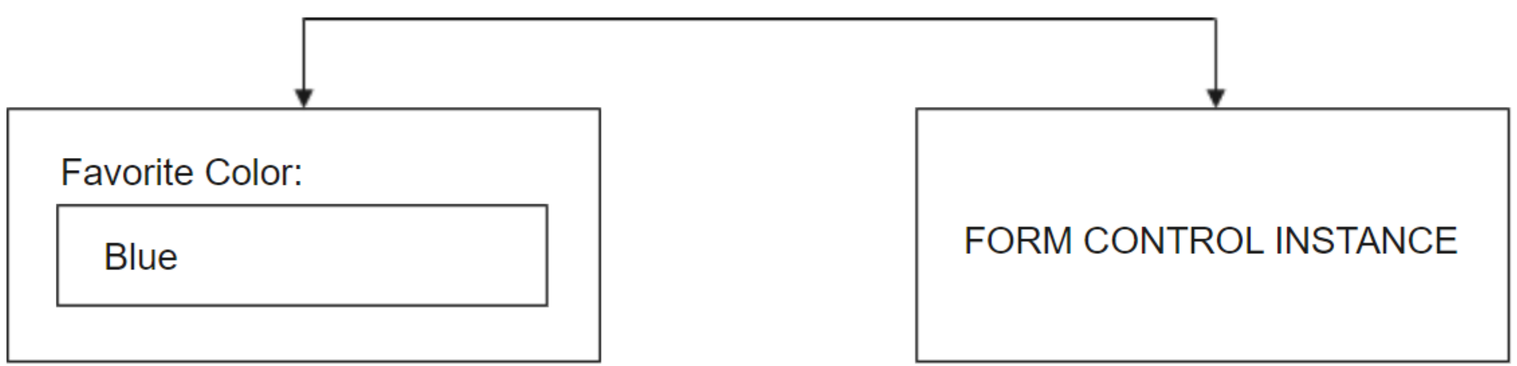
Set up the Form Model

REACTIVE FORMS

D

FORM CONTROL DIRECTIVE

Direct access to `FormControl` instance after link is created by `FormControlDirective`



Slide Note

Grouping Form Controls

- Forms typically contain several related controls.
- Reactive forms provide two ways of grouping multiple related controls into a single input form:
  - A form *group* defines a form with a fixed set of controls that can be managed together. It is also possible to nest form groups to create more complex forms.
  - A form *array* defines a dynamic form, where controls are added and removed at run time. It is also possible to nest form arrays to create more complex forms.

Slide Note

Form Group

- A Form control instance gives control over a single input field.
- A form group instance tracks the form state of a group of form control instances (for example, a form). Each control in a form group instance is tracked by name when creating the form group.
- To add a form group to a component, take the following steps:

1. Create a FormGroup instance.
2. Associate the FormGroup model and view.
3. Save the form data.

Slide Note

```
import { Component, OnInit }
from '@angular/core';
import { FormControl }
from '@angular/forms';
@Component({
selector: 'app-login',
template: `<form [formGroup]="userForm"
(ngSubmit)="onSubmit()">
<label for="name">First Name:</label>
<input type="text" id="firstName"
[formControl]="firstName">
<label for="name">Last Name:</label>
<input type="text"
id="lastName" [formControl]="lastName">`,
styleUrls: ['./user.component.css']
})
export class UserComponent {
   userForm = new FormGroup({
     firstName = new FormControl('');
     lastName = new FormControl('');
   })
}
```

Adding Form Group and Form Control

- With reactive forms, the form model is defined directly in the component class.
- Create a property named `userForm` and set it to a new form group instance.
- In the form group constructor, provide an object of named keys mapped to their control.
- Declare 2 form control instances: `firstName` and `lastName`.
- The constructor of `FormControl` is used to set its initial value (empty string in the sample code).
- With this `FormControl` instance, it is possible to listen for, update, and validate the state of the form input.
- The `[formControl]` directive links the explicitly created `FormControl` instance to a specific form element in the view, using an internal value accessor.
- The form control and DOM element communicate with each other: the view reflects changes in the model, and the model reflects changes in the view.

Slide Note

Displaying Form Value

We can display the value in the following ways:

- Through the `valueChanges` observable, where changes in the form's value are listened to in the component class using the `subscribe()` method.
- With the `value` property that gives you a snapshot of the current value.

```ts
this.userForm.valueChanges.subscribe(
value => {
  console.log('form value changed')
  console.log(value)
})
```

profile.component.ts

```html
<!--display the current value using
interpolation in the template -->
<p> Value:{{ userForm.value }}</p>
```

profile.component.html

Slide Note

Replacing Form Value

- Reactive forms have methods to change a control's value programmatically, which provides the flexibility to update the value without user interaction.
- A form group instance provides a `setValue()` method. It updates the value of the form controls and validates the structure of the value provided against the controls' structure.
- The form model is the source of truth for the control. Hence, when the button is clicked, the value of the input is changed within the component class, overriding its current value.

```ts
// update form value using setValue() method
updateProfile() {
  this.userForm.setValue({
     firstName: 'Paul',
     lastName: 'Harris'
   });
}
```

profile.component.ts

```html
<button (click)="updateProfile()">
update Profile
</button>
```

profile.component.html

Slide Note

Create a Profile Form Using Reactive Approach

Create a simple user profile form using Angular reactive forms which enables to add a new user with details like firstName and lastName.

Use the FormGroup and FormControl classes to programmatically create the form model in the component class.

Click here for the demo solution.

DEMO

Slide Note

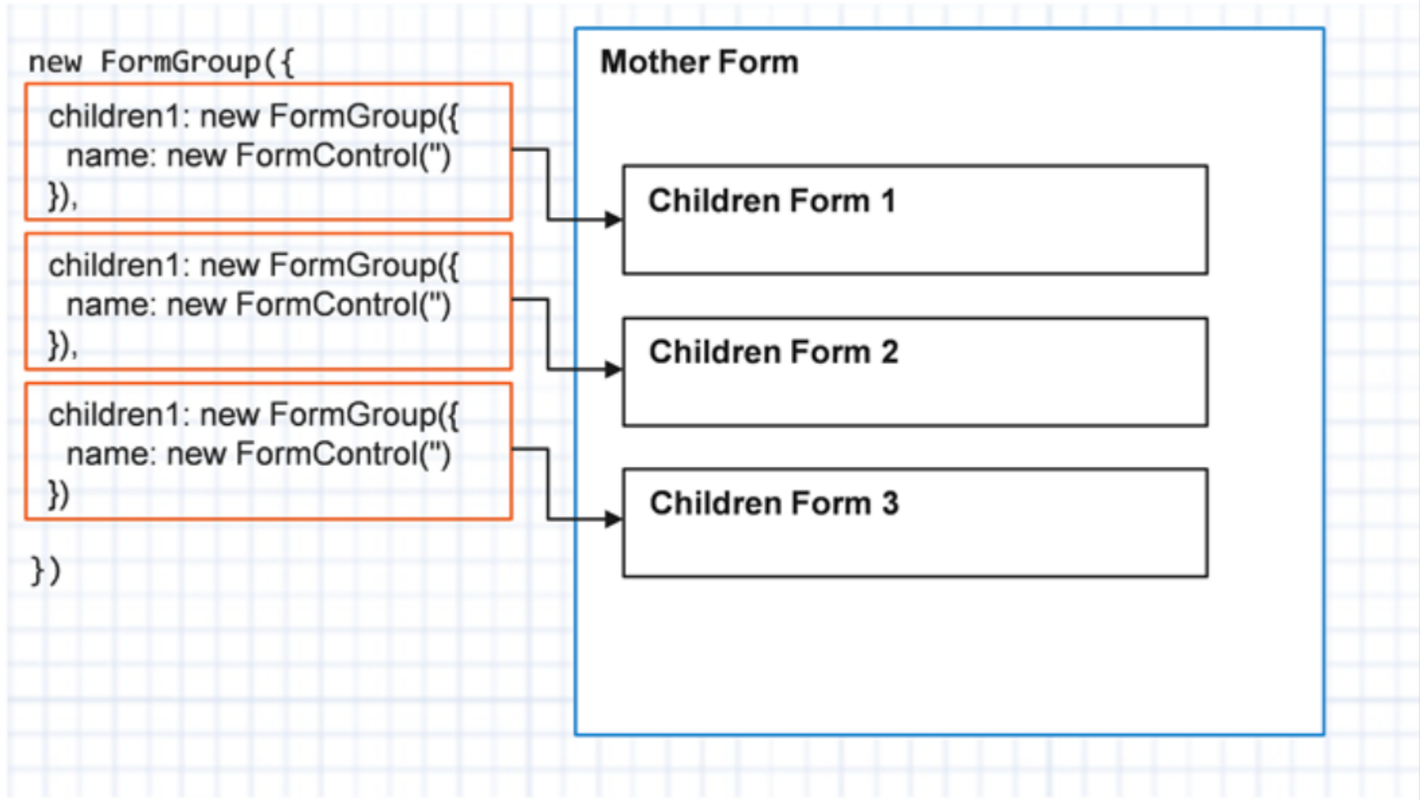
The image here shows how multiple form groups can be nested to create a single form.

Nested Form Group

A nested form group instance allows you to break large forms' groups into smaller, more manageable ones.

To make more complex forms, use the following steps:

1. Create a nested group.
2. Group the nested form group in the template.



Slide Note

Updating Parts of the Model in Form Group

- When updating the value for a form group instance that contains multiple controls, you may only want to update parts of the model.
- There are two ways to update the model value:
  - Use the `setValue()` method to set a new value for the individual control. The `setValue()` method strictly adheres to the structure of the form group and replaces the entire value for the control.
  - Use the `patchValue()` method to replace any properties defined in the object that have changed in the form model.
- The strict checks of the `setValue()` method help catch nesting errors in complex forms, while `patchValue()` fails silently on those errors.

Slide Note

Modify the User Profile Form With Nested Form Groups

Modify the user profile form by adding an address property to the form.

Use the nested form group for the address property, which has street, city, state, and zip form controls.

Click here for the demo solution.

DEMO

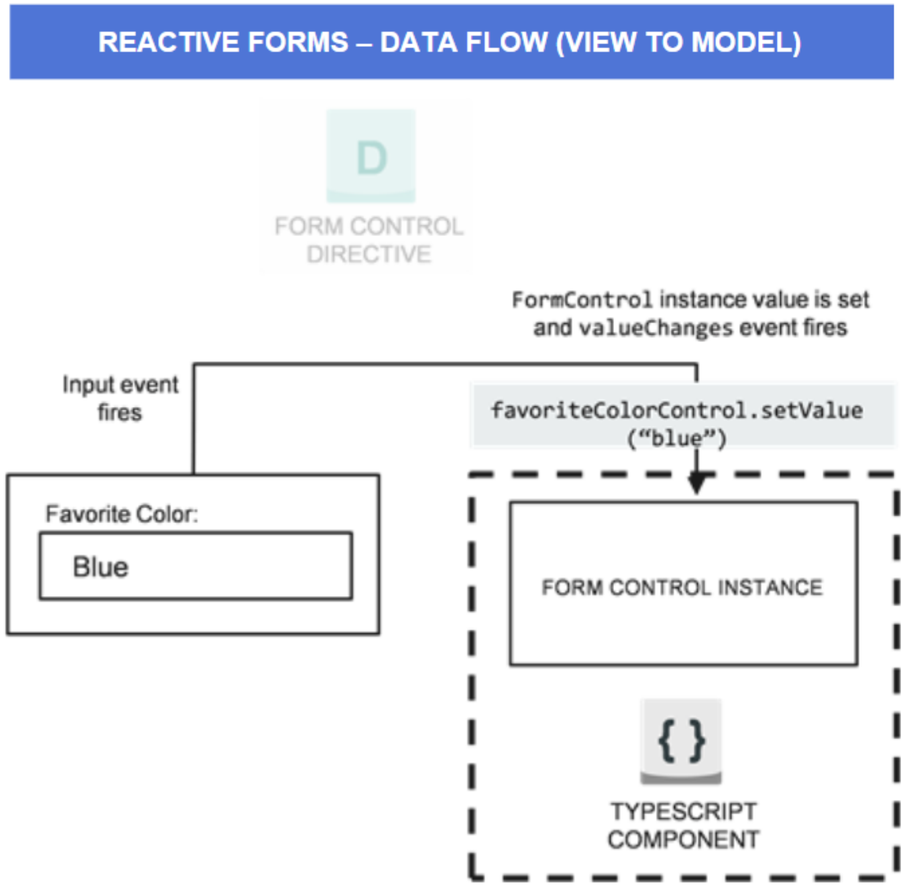
Slide Note

Synchronous Data Flow Between the View and the Data Model in Reactive Forms

Slide Note

The view-to-model diagram shows how data flows when an input field's value is changed from the view, which is explained in the next slide.

View-to-Model in Reactive Forms



Slide Note

Data Flow From View-to-Model

In reactive forms, each form element in the view is directly linked to the form model (a `FormControl` instance).

Updates from the view to the model and from the model to the view are synchronous and do not depend on how the UI is rendered.

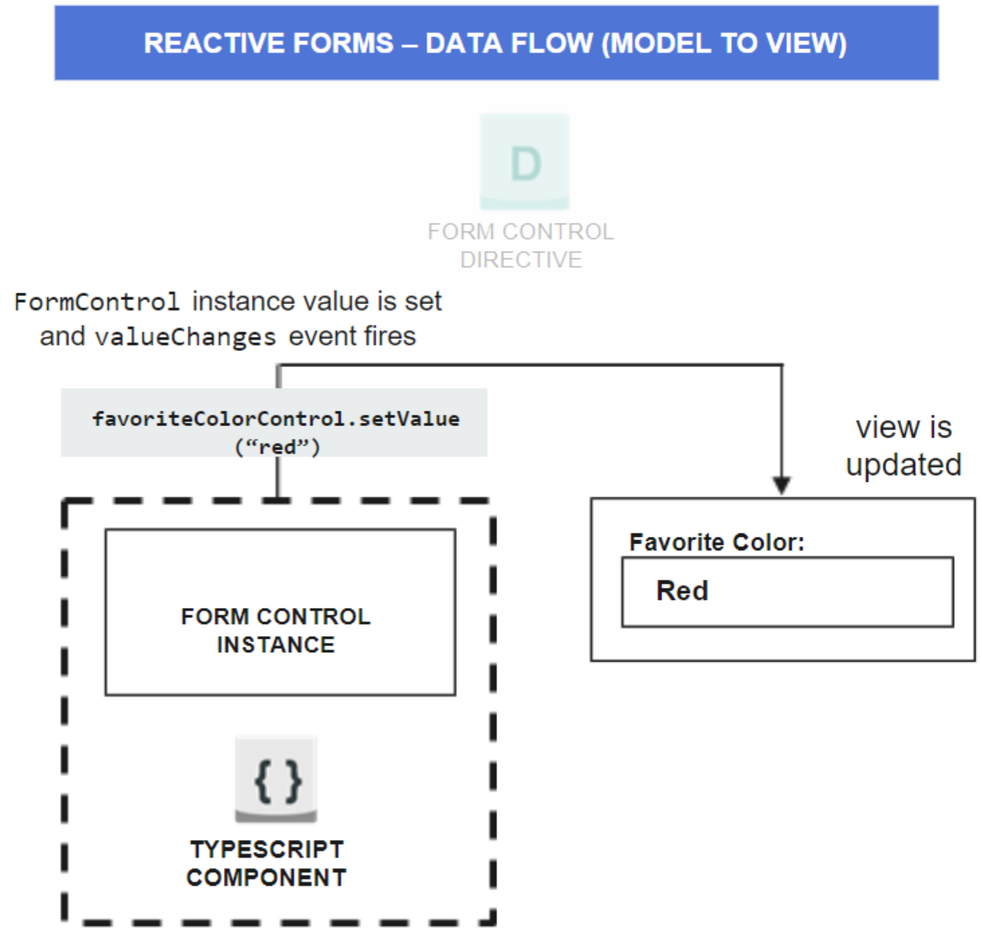
The following steps describe how data flows when an input field's value is changed from the view:

1. The user types a value into the input element; for example, the favorite color *Blue*.
2. The form input element emits an "input" event with the latest value.
3. The control value accessor listening for events on the form input element immediately relays the new value to the `FormControl` instance.
4. The `FormControl` instance emits the new value through the `valueChanges` observable.
5. Any subscribers to the `valueChanges` observable receive the new value.

Slide Note

The model-to-view diagram shows how a programmatic change to the model is propagated to the view, which is explained in the next slide.

Model-to-View in Reactive Forms



Slide Note

Data Flow From Model-to-View

The following steps describe how a programmatic change to the model is propagated to the view:

1. The user calls the `favoriteColorControl.setValue()` method, which updates the `FormControl` value.
2. The `FormControl` instance emits the new value through the `valueChanges` observable.
3. Any subscribers to the `valueChanges` observable receive the new value.
4. The control value accessor on the form input element updates the element with the new value.

Slide Note

Generating Controls Using FormBuilder Service

Slide Note

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';
@Component({
selector: 'app-profile-editor',
templateUrl: './profile-editor.component.html',
styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });
constructor(private fb: FormBuilder) { }
```

Form Builder

- Manually creating form controls with multiple forms can become repetitive.
- Angular provides the `FormBuilder` service, which offers much easier methods for generating form controls.
- Steps for using the `FormBuilder` service:
  1. Import the `FormBuilder` class.
  2. Inject the `FormBuilder` service by adding it to the component's constructor.
  3. Generate form contents using its factory methods: `control()`, `group()` and `array()`.
- In the example, the `group()` method with the same object is used to define the properties in the model.
- The value for each control name is an array. The initial value of it is given as the first item in the array.

Slide Note

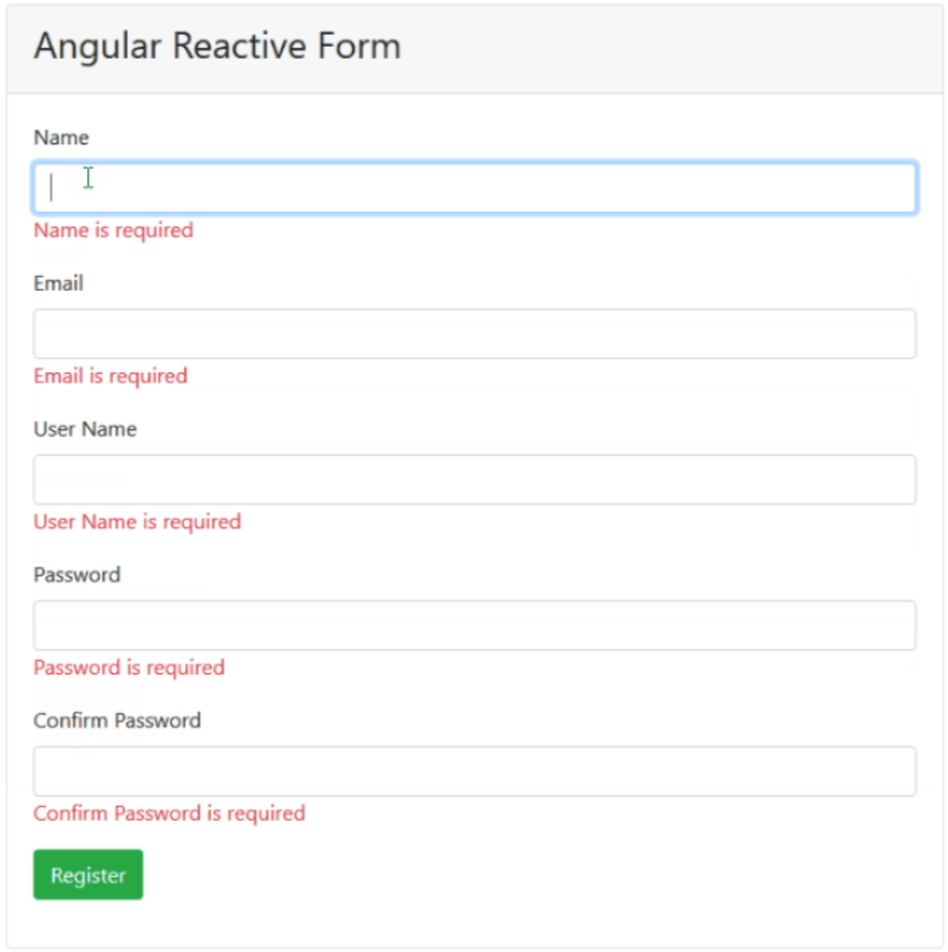
☐ Did you notice in the form that values are saved automatically, and we can undo the changes to an earlier version?

Form Input Validation in Reactive Forms

Slide Note

The video above is an example of a reactive form, where form fields are validated as they are entered and validation error messages are displayed immediately for each form field.

Reactive Form Validation



Slide Note

Validating Input Using Functions in Reactive Forms

- In the reactive form, the source of truth is the component class.
- Instead of adding validators through attributes in the template, we add validator functions directly to the form control model in the component class.
- Angular then calls these functions whenever the value of the control changes.
- Validator functions can be either synchronous or asynchronous:
  - **Synchronous validators**: Synchronous functions take a control instance and immediately return either a set of validation errors or null.
    - These are passed as the second argument while instantiating a FormControl.
  - **Asynchronous validators**: Asynchronous functions take a control instance and return a Promise or Observable that later emits a set of validation errors or null.
    - These are passed as the third argument while instantiating a FormControl.

Slide Note

In reactive form, getter methods are added to access any form control through the get method on its parent group.

Built-in Validator Functions

- The same built-in validators that are available as attributes in template-driven forms, such as `required` and `minlength`, are all available to use as functions from the validators class.
- Form controls support multiple validators by passing the functions as an array.
- Steps to add form validation:
  1. Import a validator function in the form component. These functions receive a control to validate against and return an error object or a null value based on the validation check.
  2. Add the validators static method like `required`, `minLength` as the second item in the array for the form control.
  3. Add logic to handle the validation status, like disabling the submit button when the form is invalid or simply displaying the form status.

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

export class ProfileComponent {
  profileForm = this.fb.group({
    firstName: ['', [Validators.required,
Validators.minLength(3)]],
    lastName: ['',Validators.required],
    address: this.fb.group({
        street: [''],
        city: [''],
        state: [''],
        zip: ['']
    })
});
  constructor(private fb: FormBuilder) { }
  get firstName(){return profileForm.get(
'firstName');}
}
```

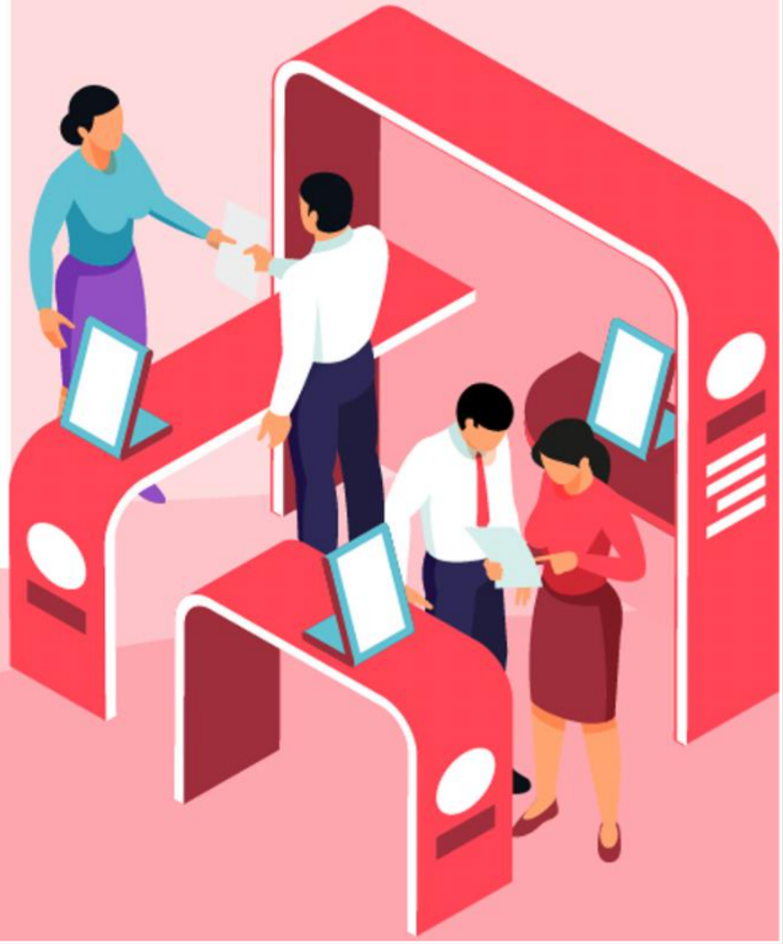
```
</p>Form Status:
{{ profileForm.status }}</p>
```

Slide Note

Add Built-in Validators to the User Profile Form

Add built-in validators like required, minlength, and pattern to the user profile form to validate various form control elements. Display custom error messages for each invalid status of form controls.

Click here for the demo solution.



DEMO

Slide Note

Note: Alternatively, the custom validator function can be created inside the component class instead of creating it in a separate file when it is not reused.

```
import {ValidatorFn, ValidationErrors,
AbstractControl} from '@angular/forms';
export function
forbiddenNameValidator(nameRe: RegExp):
ValidatorFn {
    return (control: AbstractControl):
ValidationErrors | null => {
       const forbidden =
nameRe.test(control.value);
       return forbidden ?
{forbiddenName:  {value: control.value}} :
null;
   };
 }
```

Defining Custom Validators

- The built-in validators don't always match the exact use case of your application, and it is required to create a custom validator.
- The example code creates a custom validator called `forbiddenNameValidator`.
- The function is a factory that takes a regular expression to detect a *specific* forbidden name and returns a validator function.
- The validator function takes a form control object. It returns either null (if the value of the control is valid) *or* a validation error object.
- The validation error object has a property whose key is "`forbiddenName`" and value is an arbitrary dictionary of values inserted as an error message.

Slide Note

Adding a Custom Validator to Reactive Forms

- In reactive forms, add a custom validator by passing the function directly to the FormControl.
- Custom validators can be added to the FormControl along with the built-in validators in an array.
- In the template, error messages can be provided for the custom validation errors.

```
profileForm = this.fb.group({
 firstName: ['', [Validators.required,
forbiddenNameValidator(/bob/i)],
 lastName: ['', Validators.required]
});
```

profile.component.ts

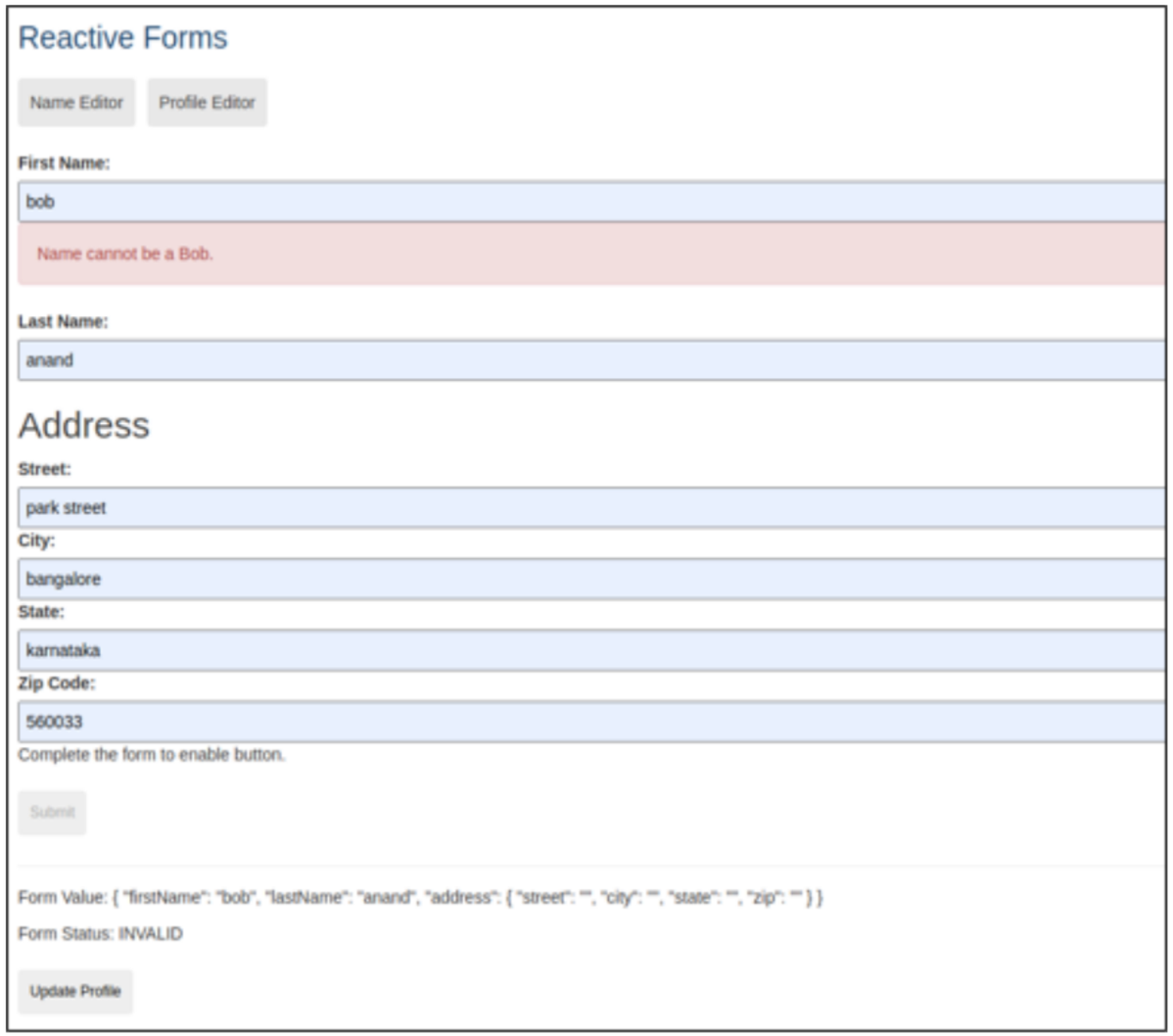
```
<input type="text" id="name"
class="form-control"
formControlName="name" required>
<div
*ngIf="name.errors?.['forbiddenName']">
Name cannot be Bob.
</div>
```

profile.component.html

Slide Note

The above image shows the validation error message for the first name field when the user fills the registration form. The first name field is validated with a custom validator function which rejects any name that the configuring regular expression matches.

Custom Validator Error Messages



Slide Note

Custom Validators in Reactive Forms

Add a custom validator called `mustMatchValidator` to the profile form. Here, the validator checks whether the password and confirm password values are the same or not.

Click here for the demo solution.

DEMO

Slide Note

Quick Check

Which one is **not** a feature of reactive forms in Angular?

1. They abstract away the underlying form API
2. Allow the creation of dynamic fields in the form
3. Suitable for complex scenarios
4. Form model is explicit, rather than implicit

Slide Note

Quick Check: Solution

Which one is **not** a feature of reactive forms in Angular?

1. **They abstract away the underlying form API**
2. Allow creation of dynamic fields in the form
3. Suitable for complex scenarios
4. Form model is explicit, rather than implicit

Explanation:
Option 1 is correct: Reactive forms provide direct access to the underlying API. Only template-driven forms abstract away the underlying form API.