CSM
breaking thought barriers

Angular Training

Session -18

Outlines

CSM
breaking thought barriers

- RF
- HTTP CLIENT
- Angular LifeCycle Hook

Angular Forms

Why we need Forms?

Forms are the main building blocks of any type of application. When we use forms for login, registration, submission.

Forms are really very very important to collect the data from the users. Often, each website contains forms to collect the user data.

You can use forms to login, submit a help request, place an order, book a flight, schedule a meeting, and perform other countless data entry tasks.

What are Angular Forms?

The Angular Framework, provides two different ways to collect and validate the data from a user.

They are :

1. Template-Driven Forms
2. Model-Driven Forms (Reactive Forms)

Template Driven Forms in Angular:

- Template Driven Forms are simple forms which can be used to develop forms.

- These are called Template Driven as everything that we are going to use in an application is defined into the template that we are defining along with the component.

Features of Template Driven Forms:

- ✓ Easy to use.
- ✓ Suitable for simple scenarios and fail for complex scenarios.
- ✓ Similar to Angular 1.0 (Angular JS)
- ✓ Two way data binding using NgModule syntax.
- ✓ Minimal Component code
- ✓ Automatic track of the form and its data.
- ✓ Unit testing is another challenge

Model-Driven Forms (Reactive Forms) in Angular

In a model driven approach, the model which is created in the .ts file is responsible for handling all the user interactions and validations.

For this, first, we need to create the model using Angular's inbuilt classes like formGroup and formControl and then we need to bind that model to the HTML form.

This approach uses the Reactive forms for developing the forms which favor the explicit management of data between the UI (User Interface) and the Model.

With this approach, we create the tree of Angular Form Controls and bind them in the Native Form Controls.

As we create the form controls directly in the component, it makes it a bit easier to push the data between the data models and the UI elements.

In order to use Reactive Forms, you need to import ReactiveFormsModule into the applications root module i.e. app.module.ts file.

Features of Reactive Forms:

- ✓ More flexible, but need a lot of practice
- ✓ Handles any complex scenarios.
- ✓ No data binding is done (Immutable data model preferred by most developers).
- ✓ More component code and less HTML Markup.
- ✓ Easier unit testing.
- ✓ Reactive transformations can be made possible such as
- ✓ Handling a event based on a denounce time.
- ✓ Handling events when the components are distinct until changed.
- ✓ Adding elements dynamically.

Note: They both are two different approaches, so we can use whichever suits our needs the most. we can use both in the same application.

Template-driven forms	Reactive forms
- Rely on directives in the template to create and manipulate the underlying object model. - They are useful for adding a simple form to an app, such as an email list signup form. - They're straightforward to add to an app, but they don't scale as well as reactive forms. - If you have very basic form requirements and logic that can be managed solely in the template, template-driven forms could be a good fit.	- Provide direct, explicit access to the underlying form's object model. - They are more robust: they're more scalable, reusable, and testable. - If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

Key differences

The following table summarizes the key differences between reactive and 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

Angular Template Driven Forms

Build a template-driven form

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

DIRECTIVES	DETAILS
`NgModel`	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.
`NgForm`	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.
`NgModelGroup`	Creates and binds a `FormGroup` instance to a DOM element.

Common form foundation classes

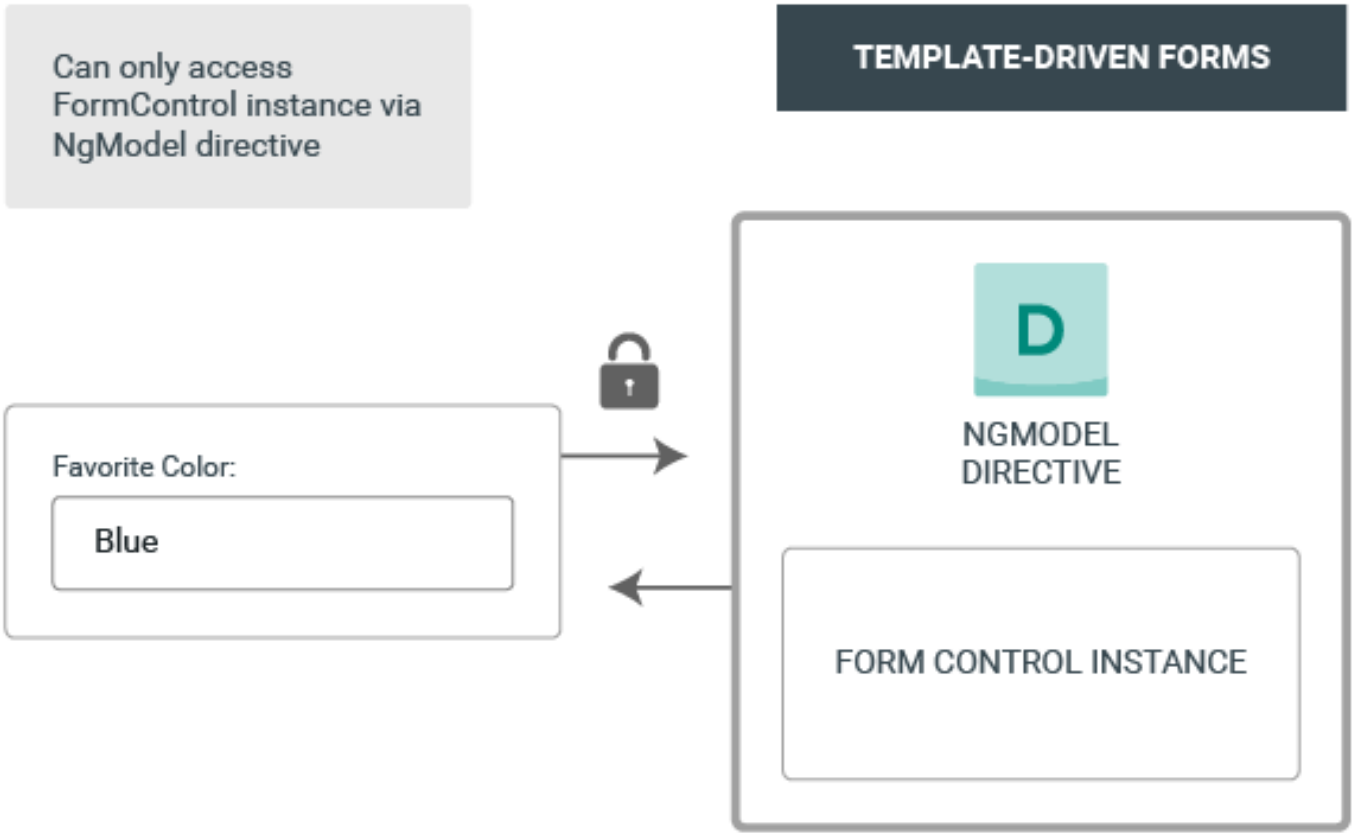
Both reactive and template-driven forms are built on the following base classes.

BASE CLASSES	DETAILS
`FormControl`	Tracks the value and validation status of an individual form control.
`FormGroup`	Tracks the same values and status for a collection of form controls.
`FormArray`	Tracks the same values and status for an array of form controls.
`ControlValueAccessor`	Creates a bridge between Angular `FormControl` instances and built-in DOM elements.

Setup in template-driven forms :

- In template-driven forms, the form model is implicit, rather than explicit.
- The directive NgModel creates and manages a FormControl instance for a given form element.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `
})
export class FavoriteColorComponent {
  favoriteColor = '';
}
```

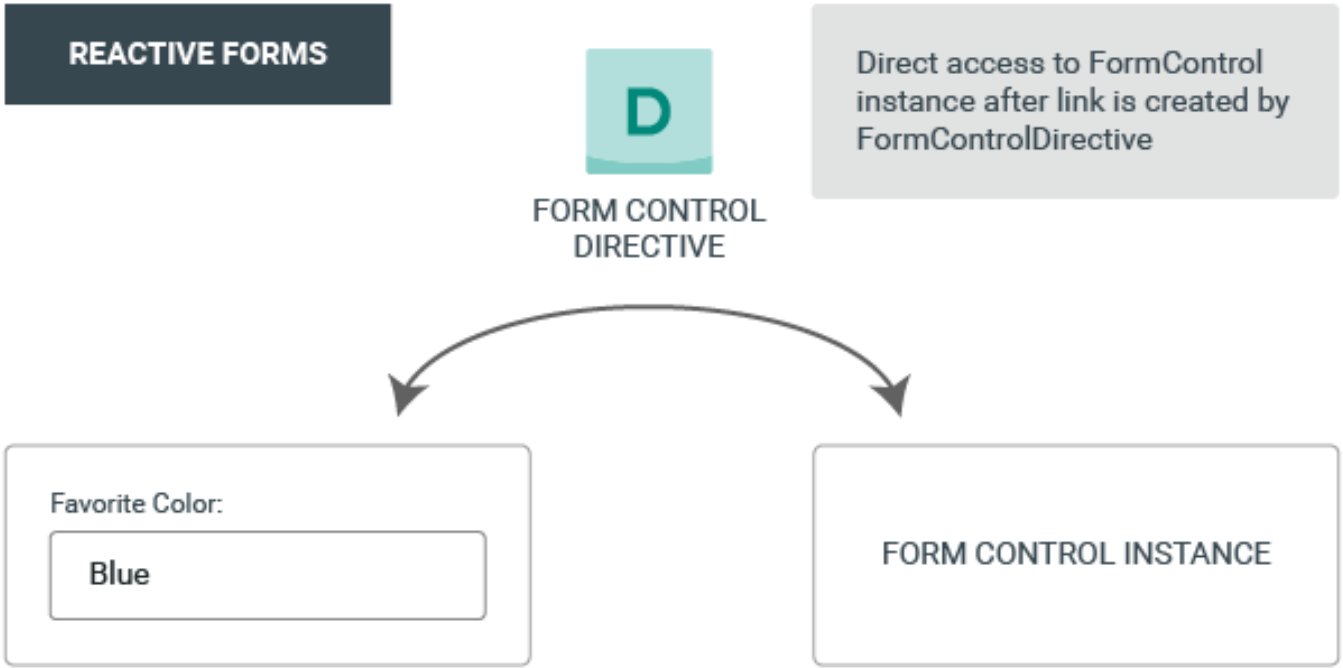

Can only access FormControl instance via NgModel directive
TEMPLATE-DRIVEN FORMS
D
NGMODEL DIRECTIVE
Favorite Color:
Blue
FORM CONTROL INSTANCE

Setup in reactive forms

- With reactive forms, we define the form model directly in the component class.
- The **[formControl]** directive links the explicitly created FormControl instance to a specific form element in the view, using an internal value accessor.

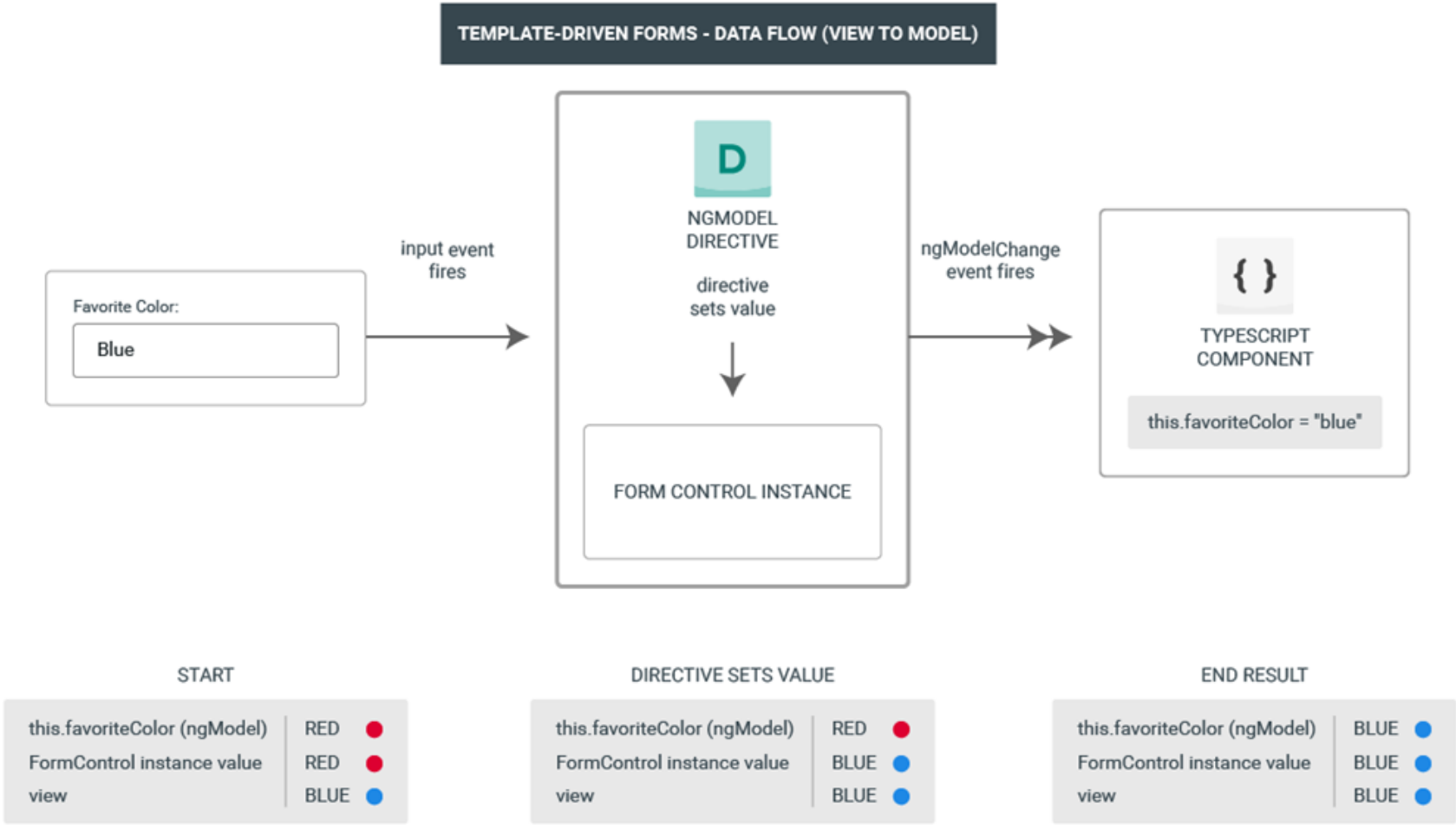
```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text" [formControl]="favoriteColorControl">
  `
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```

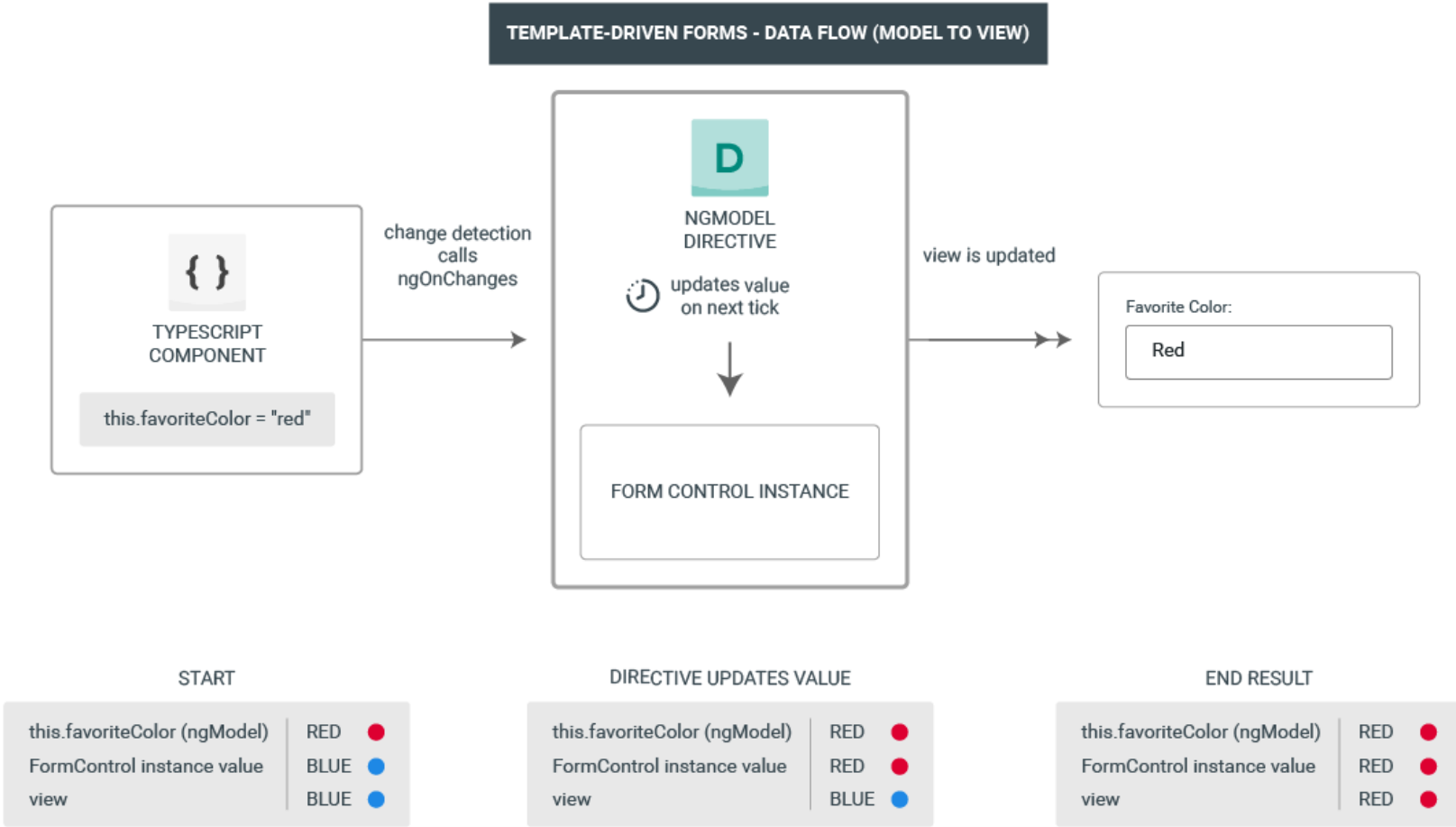

REACTIVE FORMS
D
FORM CONTROL DIRECTIVE
Direct access to FormControl instance after link is created by FormControlDirective
Favorite Color:
Blue
FORM CONTROL INSTANCE

Data flow in template-driven forms

In template-driven forms, each form element is linked to a directive that manages the form model internally.


TEMPLATE-DRIVEN FORMS - DATA FLOW (VIEW TO MODEL)
Favorite Color:
Blue
input event fires
D
NGMODEL DIRECTIVE
directive sets value
FORM CONTROL INSTANCE
ngModelChange event fires
{ }
TYPESCRIPT COMPONENT
this.favoriteColor = "blue"
START
this.favoriteColor (ngModel) RED
FormControl instance value RED
view BLUE
DIRECTIVE SETS VALUE
this.favoriteColor (ngModel) RED
FormControl instance value BLUE
view BLUE
END RESULT
this.favoriteColor (ngModel) BLUE
FormControl instance value BLUE
view BLUE

1. The user types Blue into the input element.
2. The input element emits an "input" event with the value Blue.
3. The control value accessor attached to the input triggers the setValue() method on 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.
6. The control value accessor also calls the NgModel.viewToModelUpdate() method which emits an ngModelChange event.
7. Because the component template uses two-way data binding for the favoriteColor property, the favoriteColor property in the component is updated to the value emitted by the ngModelChange event (Blue).


TEMPLATE-DRIVEN FORMS - DATA FLOW (MODEL TO VIEW)
{ }
TYPESCRIPT COMPONENT
this.favoriteColor = "red"
change detection calls ngOnChanges
D
NGMODEL DIRECTIVE
updates value on next tick
FORM CONTROL INSTANCE
view is updated
Favorite Color:
Red
START
this.favoriteColor (ngModel) RED
FormControl instance value BLUE
view BLUE
DIRECTIVE UPDATES VALUE
this.favoriteColor (ngModel) RED
FormControl instance value RED
view BLUE
END RESULT
this.favoriteColor (ngModel) RED
FormControl instance value RED
view RED

1. The favoriteColor value is updated in the component.
2. Change detection begins.
3. During change detection, the ngOnChanges lifecycle hook is called on the NgModel directive instance because the value of one of its inputs has changed.
4. The ngOnChanges() method queues an async task to set the value for the internal FormControl instance.
5. Change detection completes.
6. On the next tick, the task to set the FormControl instance value is executed.
7. The FormControl instance emits the latest value through the valueChanges observable.
8. Any subscribers to the valueChanges observable receive the new value.
9. The control value accessor updates the form input element in the view with the latest favoriteColor value.

1. Build the Basic Form
2. Bind form controls to data properties using ngModel Directive and two-way-data-binding
3. Track I/P Validity
4. Response to button click by adding data.
5. Handle form submission ngSubmit
6. Bind I/P Controls to data Properties
7. Access the Overall form status
8. Track Form States
9. Track Control States
10. Respond to Form Submission

Track control states

STATES	CLASS IF TRUE	CLASS IF FALSE
The control has been visited.	`ng-touched`	`ng-untouched`
The control's value has changed.	`ng-dirty`	`ng-pristine`
The control's value is valid.	`ng-valid`	`ng-invalid`

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

Reactive forms

Reactive Forms in Angular provide a powerful way to manage and validate form inputs in a more reactive and flexible manner compared to template-driven forms.

They are built on top of the RxJS library and offer a declarative approach to working with forms.

Steps

1. **Import Required Modules**

2. **Create the Reactive Form**

3. **Create the Form Template**

4. **Display Form Validation Errors**

5. **Apply Styling (Optional)**

6. **Use the Reactive Form Component**

1. Import Required Modules

```
import { ReactiveFormsModule } from
'@angular/forms';

@NgModule({
  declarations: [/* ... */],
  imports: [
    /* ... */
    ReactiveFormsModule, // Add this line
  ],
  bootstrap: [/* ... */],
})
export class AppModule {}
```

2. Create the Reactive Form

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
@Component({
  selector: 'app-user-form',
  templateUrl: './user-form.component.html',
  styleUrls: ['./user-form.component.css'],
})
export class UserFormComponent implements OnInit {
  userForm: FormGroup; // Declare a FormGroup variable
  constructor(private fb: FormBuilder) {
    // Initialize the form using FormBuilder
    this.userForm = this.fb.group({
      // Define form controls with initial values and validators
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      age: [null, [Validators.required, Validators.min(18)]],
    });
  }
```

```
ngOnInit(): void {}
  // Implement form submission logic
  onSubmit() {
    if (this.userForm.valid) {
      console.log( this.userForm.value ); }
  }
}
```

3. Create the Form Template

```
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="firstName">First Name</label>
    <input type="text" id="firstName" formControlName="firstName" />
  </div>
  <div class="form-group">
    <label for="lastName">Last Name</label>
    <input type="text" id="lastName" formControlName="lastName" />
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input type="email" id="email" formControlName="email" />
  </div>
  <div class="form-group">
    <label for="age">Age</label>
    <input type="number" id="age" formControlName="age" />
  </div>
  <button type="submit" [disabled]="userForm.invalid">Submit</button>
</form>
```

4. **Display Form Validation Errors**

```
<div class="form-group">
  <label for="firstName">First Name</label>
  <input type="text" id="firstName" formControlName="firstName" />
  <div *ngIf="userForm.get('firstName').hasError('required')" class="error">
    First Name is required.
  </div>
</div>
<!-- Repeat this for other fields -->
```

5. Apply Styling (Optional)

```
.form-group {
  margin-bottom: 15px;
}

label {
  display: block;
  font-weight: bold;
}

.error {
  color: red;
  font-size: 12px;
}
```

6. Use the Reactive Form Component

```
<app-user-form></app-user-form>
```

HTTP Client


http://

Outlines :

- Features of HTTP
- Using HttpClient methods
- Working with HTTP

Angular provides a simplistic HTTP API for performing HTTP operations in angular applications.

The front-end applications need to communicate with a server over the HTTP protocol to download data, upload data, and access other back-end services.

Features of HTTP

It offers features like:

- Error handling
- Request and response interception.
- Typed response objects
- Stateless
- Text-Based
- URI (Uniform Resource Identifier)
- Methods
- Headers
- Status Codes

HTTP Methods :

In Angular, you can use the HttpClient service to make various HTTP requests to interact with remote servers or APIs.

HttpClient provides methods for different HTTP methods like GET, POST, PUT, DELETE, and more.

1. GET Request:

To make a GET request using HttpClient, you can use the get() method.

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }

ngOnInit(): void {
  this.http.get('https://api.example.com/data').subscribe((data) => {
    // Handle the response data
    console.log(data);
  });
}
```

2. POST Request:

To make a POST request with data, you can use the post() method.

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }

submitData(data: any): void {
  this.http.post('https://api.example.com/submit', data).subscribe((response) => {
    // Handle the response
    console.log(response);
  });
}
```

3.**PUT Request:**

For PUT requests to update data on the server, use the put() method:

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }

updateData(data: any): void {
  this.http.put('https://api.example.com/update', data).subscribe((response) => {
    // Handle the response
    console.log(response);
  });
}
```

4. **DELETE Request:**

To send a DELETE request to remove data on the server, use the delete() method:

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }

deleteData(id: number): void {
  this.http.delete(`https://api.example.com/delete/${id}`).subscribe((response) => {
    // Handle the response
    console.log(response);
  });
}
```

5.Setting Headers and Query Parameters:

You can set custom headers and **query parameters** using the HttpHeaders and **HttpParams** classes:

```
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';
constructor(private http: HttpClient) { }
getDataWithHeadersAndParams(): void {
  const headers = new HttpHeaders({
    'Authorization': 'Bearer your-token',
    'Custom-Header': 'custom-value'
  });
  const params = new HttpParams()
    .set('param1', 'value1')
    .set('param2', 'value2');
  const options = { headers: headers, params: params };
  this.http.get('https://api.example.com/data', options).subscribe((data) => {
    // Handle the response data
    console.log(data);
  });
}
```

Working with HTTP in angular

1. Import HttpClientModule

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [HttpClientModule],
  // ...
})
export class AppModule { }
```

2. Inject HttpClient

```
import { HttpClient } from '@angular/common/http';
constructor(private http: HttpClient) { }
```

3. Making HTTP Requests

```
// Making a GET request
this.http.get('/api/data').subscribe((data) => {
  // Handle the response data
  console.log(data);
});
```

4. **Handling Responses:**

When you make an HTTP request, you typically subscribe to the Observable returned by the HTTP method. You can then handle the response data or errors in the subscription's callback.

```
this.http.get('/api/data').subscribe(
  (data) => {
    // Handle successful response
    console.log(data);
  },
  (error) => {
    // Handle errors
    console.error('An error occurred:', error);
  }
);
```

5. **Sending Data in POST Requests:**

```
const postData = { name: 'John', age: 30 };

this.http.post('/api/postData', postData).subscribe(
  (response) => {
    // Handle successful response
    console.log(response);
  },
  (error) => {
    // Handle errors
    console.error('An error occurred:', error);
  }
);
```

6. Handling Headers and Options:

You can also set HTTP headers and options for your requests. This can be done using the HttpHeaders class and the HttpParams class for query parameters.

```
const headers = new HttpHeaders({
  'Content-Type': 'application/json',
  'Authorization': 'Bearer your-token-here'
});
const options = {
  headers: headers,
  params: new HttpParams().set('param1', 'value1')
};
this.http.get('/api/data', options).subscribe((data) => {
  // Handle the response data
  console.log(data);
});
```

7. Using Observables:

Angular's HTTP client returns Observables. You can leverage RxJS operators to manipulate and transform the data received from HTTP requests.

```
import { map } from 'rxjs/operators';

this.http.get('/api/data').pipe(
  map((data) => {
    // Transform data as needed
    return data;
  })
).subscribe((transformedData) => {
  // Handle the transformed data
  console.log(transformedData);
});
```

8. Error Handling and Interceptors:

You can implement error handling strategies and use interceptors to intercept HTTP requests and responses globally. This allows you to apply common error handling or headers to multiple requests.

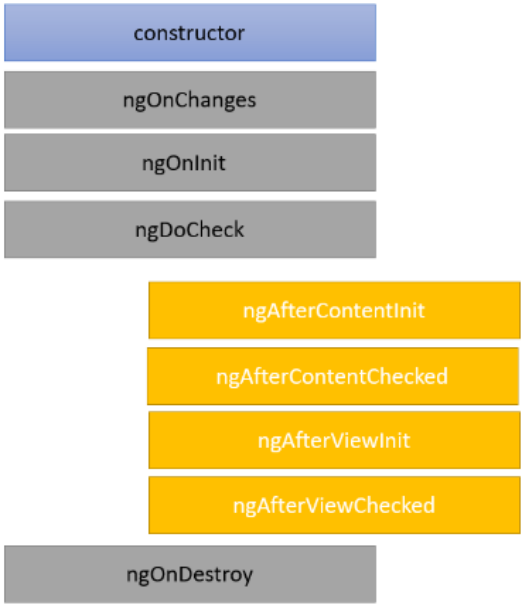
```
// Error handling example
catchError((error) => {
  console.error('An error occurred:', error);
  return throwError('Something went wrong, please try again later.');
});
```

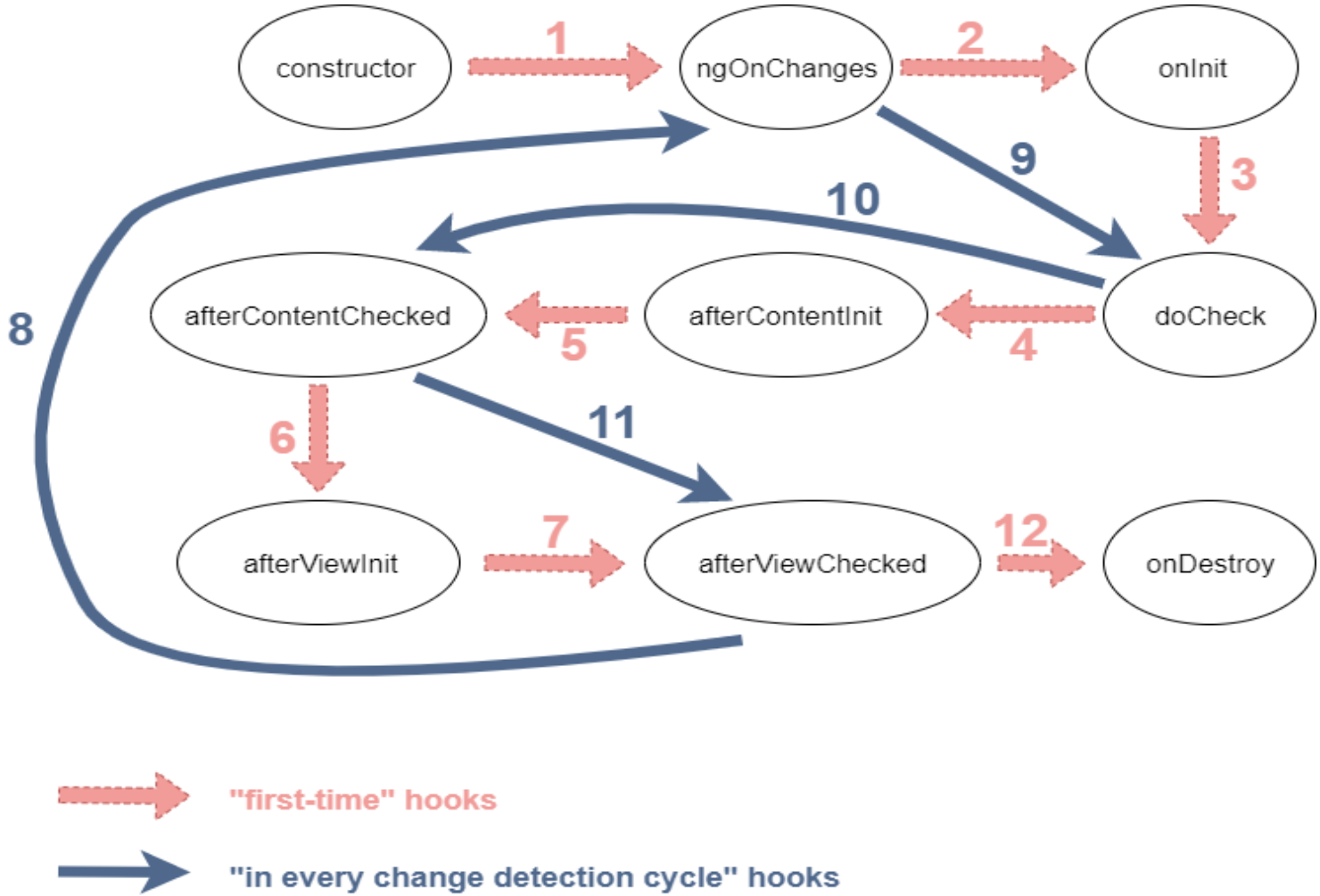
Component Lifecycle Hooks

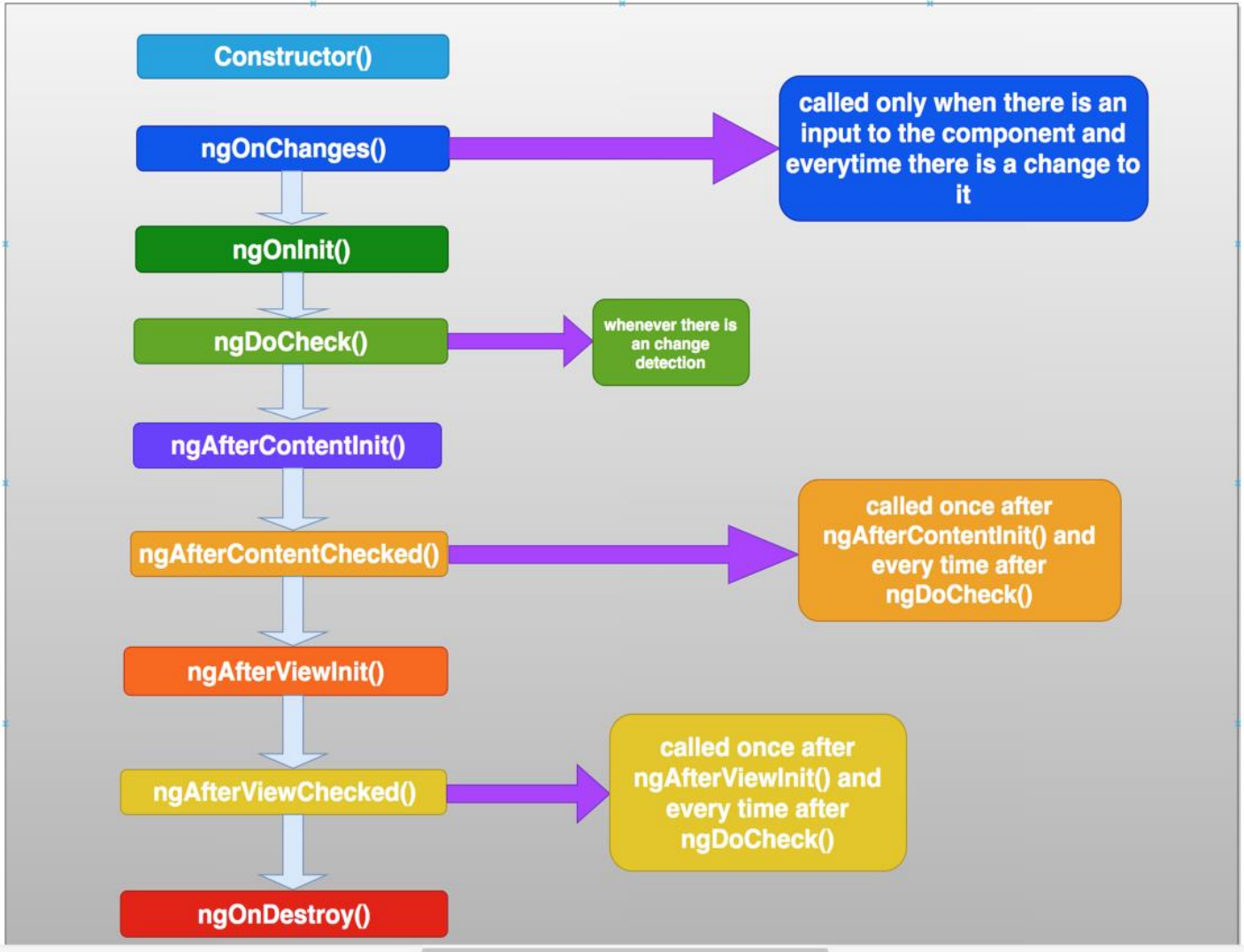
Performing CRUD Operation Using Angular

Angular provides a set of lifecycle hooks that allow you to tap into specific moments in a component's lifecycle.

These hooks provide opportunities to perform actions at different stages, such as component creation, initialization, data binding, and destruction.




constructor
1
ngOnChanges
2
onInit
3
9
10
8
afterContentChecked
5
afterContentInit
4
doCheck
6
11
afterViewInit
7
afterViewChecked
12
onDestroy
"first-time" hooks
"in every change detection cycle" hooks


Constructor()
ngOnChanges()
called only when there is an input to the component and everytime there is a change to it
ngOnInit()
ngDoCheck()
whenever there is an change detection
ngAfterContentInit()
ngAfterContentChecked()
called once after ngAfterContentInit() and every time after ngDoCheck()
ngAfterViewInit()
ngAfterViewChecked()
called once after ngAfterViewInit() and every time after ngDoCheck()
ngOnDestroy()

1.ngOnInit

ngOnInit is one of the most commonly used lifecycle hooks. It's called after Angular has initialized all data-bound properties of a directive.

```
// app.component.ts

import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  template: '<p>{{ message }}</p>',
})
export class AppComponent implements OnInit {
  message: string = '';

  ngOnInit() {
    this.message = 'Component initialized!';
  }
}
```

2. ngOnChanges:

ngOnChanges is called whenever one or more input properties of the component change.

```
// app.component.ts
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';
@Component({
  selector: 'app-root',
  template: '<p>{{ message }}</p>',
})
export class AppComponent implements OnChanges {
  @Input() inputMessage: string = '';
  message: string = '';

  ngOnChanges(changes: SimpleChanges) {
    if (changes.inputMessage) {
      this.message = `Input changed: ${changes.inputMessage.currentValue}`;
    }
  }
}
```

3.ngAfterViewInit and ngAfterViewChecked:

ngAfterViewInit is called after the component's view (HTML) has been initialized, and ngAfterViewChecked is called after every check of the component's view.

```
// app.component.ts
import { Component, AfterViewInit, AfterViewChecked } from '@angular/core';
@Component({
  selector: 'app-root',
  template: '<p>{{ message }}</p>',
})
export class AppComponent implements AfterViewInit, AfterViewChecked {
  message: string = 'View not initialized yet';

  ngAfterViewInit() {
    this.message = 'View initialized!';
  }
  ngAfterViewChecked() {
    console.log('View checked');
  }
}
```

4. ngOnDestroy:

ngOnDestroy is called just before the component is destroyed. It's useful for cleaning up resources, unsubscribing from observables, or performing any necessary cleanup:

```
// app.component.ts
import { Component, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';
@Component({
  selector: 'app-root',
  template: '<p>{{ message }}</p>',
})
export class AppComponent implements OnDestroy {
  message: string = 'Component will be destroyed';
  private subscription: Subscription | undefined;
  constructor() {
    this.subscription = new Subscription();
  }
  ngOnDestroy() {
    this.subscription?.unsubscribe();
    console.log('Component destroyed');
  }
}
```

Every time we make use of a lifecycle hook in a component, the class needs to implement that interface of the particular lifecycle hook.

```
// src/app/lifecycle-demo/lifecycle-demo.component.t
import { Component, OnInit, OnChanges, SimpleChanges, OnDestroy, AfterViewInit, AfterViewChecked } from '@angular/core';
@Component({
  selector: 'app-lifecycle-demo',
  templateUrl: './lifecycle-demo.component.html',
  styleUrls: ['./lifecycle-demo.component.css']
})
export class LifecycleDemoComponent implements OnInit, OnChanges, OnDestroy, AfterViewInit, AfterViewChecked {

  constructor() {
    console.log('Constructor called');
  }
  ngOnInit() {
    console.log('ngOnInit called');
  }
  ngOnChanges(changes: SimpleChanges) {
    console.log('ngOnChanges called', changes);
  }

  ngAfterViewInit() {
    console.log('ngAfterViewInit called');
  }

  ngAfterViewChecked() {
    console.log('ngAfterViewChecked called');
  }

  ngOnDestroy() {
    console.log('ngOnDestroy called');
  }
}
```