Reactive forms

We can build forms in Angular with a template‑driven approach using HTML and data binding, or

we can build reactive forms using a model‑driven approach by defining the form model and validation in our component code.

any type of either template driven forms or reactive forms uses behind the scend the form group

**Reactive Forms building blocks**

**FormGroup**: Tracks the value and validity state of a group of FormControl instances.

**FormControl**: Tracks the value and validation status of an individual form control.

**Creating a form control**

The following code snippet

🡪creates a single control named firstName:

**let firstName = new FormControl();**

🡪The following code snippet creates a single control named firstName and initializes it with an empty default value:

**let firstName = new FormControl('');**

🡪The following code snippet creates a single control named firstName, and initializes with default value 'Shravan':

**let firstName = new FormControl('RamaKrishna');**

***🡪A FormControl constructor accepts three arguments***

*1,an initial value,*

*2.an array of sync validators,*

*3.an array of async validators—and as you might have guessed, they are all optional.*

**Firstname = new FormControl('', [Validators.required, Validators.maxLength(15)]);**

**Input control states**

Every input control on an Angular form and the form itself maintains different states

depending on the user input and interaction with it:

//form control error list object

**let errors = firstName.errors**

// form control value is valid, it has no errors

**let isValid = firstName.valid**

// form control value is invalid, it has errors

let isInValid = firstName.invalid

//Control has been visited

**let isTouched = firstName.touched**

//Control has not been visited

let isUnTouched = firstName.untouched

//Form control's value has changed

let valueChanged = firstName.dirty

//Form control's value has not changed

**let valueNotChanged = firstName.pristine**

**FormBuilder**: Creates an AbstractControl from a user-specified configuration.

The form model retains form state, such as dirty or valid. It retains the user's entries in its value property. As the user types into any input element on the form, the value property is changed accordingly. The form model also tracks all of the FormControls and nested FormGroups within the form, along with their state and value. D

on't confuse the form model with the data model that we use with data binding.

. We use these properties from the form model any time we need to know the state or value of a FormControl or FormGroup.

This form model is the same for both template‑driven and reactive forms, but how it is created is different.

When using a template‑driven approach, we write HTML in our template for the form element, each input element, data binding, validation rules using attributes, and validation error messages. Angular automatically generates the associated form model. We can then use the form model as needed. In our component class, we define properties for the data binding. These properties represent our data model. We also implement methods for form operations, such as getting the data for display on the form and saving that data on a submit.

Two‑way data binding is a key part of template‑driven forms, as it keeps all of the data on the form in synchronization with properties in the component class.

Reactive forms shift the responsibility for creating the form model to the ***component class***. We define the form model by creating the instances of the FormGroup and FormControl building blocks in our component class

. We define the validation rules in the class. We can even handle display of validation error messages in the class. We manage the data for the form in the class. No data binding in the HTML. And the class provides methods for form processing, such as handling the submit

Example-1

**Using FormGroup**

You can use FormGroup, as follows.

First, you need to import ReactiveFormsModule from the @angular/forms module in your application module and add it to the imports array of @NgModule as following:

**import { NgModule } from '@angular/core';**

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

**import { BrowserModule } from '@angular/platform-browser';**

**@NgModule({**

**imports: [**

**BrowserModule,**

**ReactiveFormsModule**

**],**

**})**

**export class AppModule { }**

ReactiveFormsModule provides the FormControl, FormGroup and FormArray APIs.

Next, you need to create an instance of FormGroup with the instances of FormControl:

**productForm = new FormGroup({**

**reference: new FormControl(),**

**quantity: new FormControl('11')**

**});**

You can provide a default value for the control, by passing it as an argument to the FormControl.

Next, we create a <form> element in our component's template and we use [formGroup]to bind our FormGroup and formControlName directive to bind FormControl elements to HTML form controls:

<form [formGroup]="productForm">

Reference: <input formControlName="reference" placeholder="Enter reference">

Quantity: <input formControlName="quantity" placeholder="Enter quantity">

<button type="submit">Submit</button>

</form>

These are the steps of this tutorial:

* Prerequisites
* Angular Forms, Step 1 — Installing Angular CLI 9
* Step 2 — Initializing your Angular 11 Project
* Step 3 — Adding a Reactive Form
* Step 3.1 — Importing the ReactiveFormsModule
* Step 3.2 — Importing FormControl and FormGroup
* Step 3.3 — Creating the FormGroup
* Step 3.4 — Creating the HTML Form
* Step 4 — Using the FormBuilder Module
* Conclusion

**Prerequisites**

This tutorial assumes you already have Node.js and npm installed on your machine.

You also need to be familiar with TypeScript and the basics of Angular such as [components](https://www.techiediaries.com/angular-components/).

Let's see how to install Angular 11 CLI.

**Angular Forms, Step 1 — Installing Angular CLI 11**

In this step, we'll set up Angular CLI 11 in our development machine.

Angular CLI is built on top of Node.js so as mentionned before make sure you have it installed on your machine together with npm.

[Angular CLI](https://cli.angular.io/) is the official tool for initializing and working with Angular projects.

In your terminal or command prompt run the following command:

$ npm install -g @angular/cli

This will install **angular/cli v11.0.0** in our system.

That's it, you can now initialize your project using this tool.

**Step 2 — Initializing your Angular 11 Project**

Go back to your terminal and run the following commands:

$ cd ~

$ ng new angular-forms-example

The CLI will prompt you if **You would like to add Angular routing.** You can type Yes if you need [routing in your example](https://www.techiediaries.com/angular-router/) and **which stylesheet format you would like to use.** You can select [**CSS**](https://www.techiediaries.com/css-tutorial/).

Angular CLI will prepare your project, next you can navigate to your project's folder and serve your app locally using a development server as follows

$ cd angular-forms-example

$ ng serve

Your web application will be available from the <http://localhost:4200/> address.

Go to web browser and navigate to the http://localhost:4200/ address:

**Step 3 — Adding a Reactive Form**

In this step, we'll create an example HTML form. Next, we'll create a form model in the application component using the FormGroup and FormControl APIs. Finally, we'll use the formGroup, formControlName and formGroupName directives to bind our form model to our HTML form.

**Step 3.1 — Importing the ReactiveFormsModule**

Open the src/app/app.module.ts file and import the ReactiveFormsModule as follows:

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

imports: [

...

ReactiveFormsModule

],

**Step 3.2 — Importing FormControl and FormGroup, FormArray**

Next, let's import the FormControl and FormGroup classes in the src/app/app.component.ts file.

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

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'app';

}

**Step 3.3 — Creating the FormGroup**

Next, let's create an exampleForm instance of FormGroup with two firstName and lastName form controls as follows:

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

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'app';

exampleForm = new FormGroup ({

firstName: new FormControl(),

lastName: new FormControl(),

alias: new FormArray([ new FormControl("")])

});

addNewAlias() {

this.alias.push(this.fb.control(""));

}

get aliases() {

return this.exampleForm.get("alias") as FormArray;

}

}

**Step 3.4 — Creating the HTML Form**

Next, we need to create an [HTML form](https://www.techiediaries.com/html-tutorial/) in the src/app/app.component.html file:

<h1>Angular 11 Forms Example</h1>

<form [formGroup]="exampleForm">

<div class="form-group">

<label>First Name:</label>

<input class="form-control" formControlName="firstName">

<label>Last Name:</label>

<input class="form-control" formControlName="lastName">

<div formArrayName="alias">

<h3>Add alias</h3>

<button (click)="addNewAlias();" >Add another alias </button>

<div \*ngFor="let address of aliases.controls; let i=index">

<input type="text" [formControlName] = "i" >

</div>

</div>

</div>

</form>

We use the formGroup property in the <form> tag to bind the form with our exampleForm form group and we use the formControlName property to bind the <input> tags to individual form controls.

**Step 4 — Using the FormBuilder Module**

The FormBuilder service provides three factory methods:

* control(),
* group(),
* and array().

The FormBuilder helps you create reactive forms using a simple functional API for creating form controls, form groups, and form arrays.

Inside the src/app/ap.component.ts file import the FormBuilder class from the @angular/forms package as follows:

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

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

Next, inject FormBuilder in the component constructor as formBuilder

@Component({ selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css'] })

export class AppComponent {

title = 'app';

exampleForm = new FormGroup ({ firstName: new FormControl(), lastName: new FormControl()});

constructor(private formBuilder: FormBuilder) { }

}

Next add a createForm() method with the following code:

createForm() {

this.exampleForm = this.formBuilder.group({

firstName: '',

lastName: ''

});

}

Finally call the method from the constructor:

constructor(private formBuilder: FormBuilder) {

this.createForm();

}

**Conclusion**

In this tutorial, we've seen a simple example of creating a form model and bind it to the HTML <form> element using Angular FormBuilder, FormGroup and FormControl APIs.

In Reactive forms, we create the form model in the component class. First, we need to import the FormGroup, FormControl, Validators

|  |  |
| --- | --- |
|  | Step:1  import { FormGroup, FormControl, Validators } from '@angular/forms' |

When instantiating a FormGroup, pass in a collection of child controls as the first argument. The key for each child registers the name for the control

The following form model has two Form Groups. One is the top-level Form group, which we have named as reactiveForm. The other one is nested Form Group, which we have named it as address.

|  |
| --- |
| reactiveForm = new FormGroup({    firstname: new FormControl('', [Validators.required]),    lastname: new FormControl(''),    email: new FormControl(''),    address: new FormGroup({      address: new FormControl(''),      city: new FormControl(''),      state: new FormControl(''),    })  }) |

And in the Template, we use formGroup, formControlName and formGroupName directive to bind the Form to the template.

|  |
| --- |
| <form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()" novalidate>        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" formControlName="firstname">      </p>        <p>        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" formControlName="lastname">      </p>        <p>        <label for="email">Email </label>        <input type="text" id="email" name="email" formControlName="email">      </p>        <div formGroupName="address">          <p>          <label for="address">Address</label>          <input type="text" class="form-control" name="address" formControlName="address">        </p>          <p>          <label for="city">City</label>          <input type="text" class="form-control" name="city" formControlName="city">        </p>          <p>          <label for="state">State</label>          <input type="text" class="form-control" name="state" formControlName="state">        </p>        </div>        <button>Submit</button>      </form> |

**Setting Value**

We use [setValue](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/) or patchValue method of the FormGroup to set a new value for the entire FormGroup.

**SetValue**

Sets the value of the FormGroup. It accepts an object that matches the structure of the group, with control names as keys. The structure must match exactly, otherwise, it will result in an error.

setValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

|  |  |
| --- | --- |
|  | setValue() {      this.reactiveForm.setValue({      firstname: "Sachin",      lastname: "Tendulakr",      email: "sachin@gmail.com",      address: {        address: "19-A, Perry Cross Road, Bandra (West)",        city: "Mumbai",        state: "Maharatsra",      }    })  } |

You can also update the nested FormGroup separately,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | setAddress() {    this.reactiveForm.get("address").setValue({      address: "19-A, Perry Cross Road, Bandra (West)",      city: "Mumbai",      state: "Maharatsra",    })  } |

**patchValue**

Patches the value of the [FormGroup](https://angular.io/api/forms/FormGroup). It accepts an object with control names as keys and does its best to match the values to the correct controls in the group.

patchValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | patchValue() {      this.reactiveForm.patchValue({      email: "sachin@gmail.com",      address: {        state: "Maharatsra",      }    })  } |

We can Both setValue & patchValue

* onlySelf: When true, each change only affects this control and not its parent. The default is true.
* emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control value is updated. When false, no events are emitted. The configuration options are passed to the updateValueAndValidity method.

**Finding the Value**

**value**

The value returns the object with a key-value pair for each member of the Form Group. It is Readonly. To Set [Value either setValue or patchValue](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/)

value: any

|  |  |
| --- | --- |
| 1  2  3  4  5 | onSubmit() {    console.log(this.reactiveForm.value);  } |

**valueChanges**

valueChanges: Observable<any>

The angular emits the [valueChanges](https://www.tektutorialshub.com/angular/valuechanges-in-angular-forms/) event whenever the value of any of the controls in the Form Group changes. The value may change when user updates the element in the UI or programmatically through the setValue/patchValue method. We can subscribe to it as shown below

In the example below, the first valuesChanges are fired, when any of the control is changed. While the second valuesChanges event is raised only when the controls under the address form group are changed

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  0 | ngOnInit() {    this.reactiveForm.valueChanges.subscribe(x => {      console.log(x);    })    this.reactiveForm.get("address").valueChanges.subscribe(x => {      console.log(x);    })  } |

**Adding Controls Dynamically to Form Group**

We usually add controls, while initializing the FormGroup.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | reactiveForm = new FormGroup({    firstname: new FormControl('', [Validators.required]),    } |

The Forms API also allows add controls dynamically

**addControl()**

Adds a control to the FormGroup and also updates validity & validation status. If the control already exists, then ignores it

addControl(name: string, control: AbstractControl): void

|  |  |
| --- | --- |
| 1 | addControl() {      this.middleName = new FormControl('', [Validators.required]);      this.reactiveForm.addControl("middleName",this.middleName);    } |

**registerControl()**

Adds control to this FormGroup but does not update the validity & validation status. If the control already exists, then ignores it

registerControl(name: string, control: AbstractControl): AbstractControl

|  |
| --- |
| registerControl() {    this.middleName = new FormControl('', [Validators.required]);    this.reactiveForm.addControl("middleName",this.middleName);  } |

**removeControl()**

This method will remove the control with the provided name from the FormGroup.

removeControl(name: string): void

|  |
| --- |
| remodeControl() {    this.reactiveForm.removeControl("middleName");  } |

**setControl()**

Replaces the control with the provided name with the new control.

setControl(name: string, control: AbstractControl): void

|  |  |
| --- | --- |
|  | setControl() {    this.middleName = new FormControl('test', [Validators.required]);    this.reactiveForm.setControl("middleName",this.middleName);  } |

**contains()**

Check whether the control with the provided name exists or not..

contains(controlName: string): boolean

|  |  |
| --- | --- |
|  | containsControl() {    console.log(this.reactiveForm.contains("middleName"));  } |

**Control Status**

The FormGroup tracks the validation *status* of all the FormControls, which is part of the FormGroup. That also includes the status of nested FormGroup or FormArray. If any of the control becomes invalid, then the entire FormGroup becomes invalid.

The following is the list of status-related properties

**status**

status: string

The Angular runs validation checks, whenever the value of a *form control* changes. Based on the result of the validation, the FormGroup can have four possible states.

**VALID:** All the controls of the FormGroup has passed all validation checks.  
**INVALID:** At least one of the control has failed at least one validation check.  
**PENDING:** This Group is in the midst of conducting a validation check.  
**DISABLED:** This FormGroup is exempt from validation checks

|  |  |
| --- | --- |
|  | //reactive forms  this.reactiveForm.status |

**valid**

valid: boolean

A FormGroup is valid when it has passed all the validation checks and the FormGroup is not disabled

|  |  |
| --- | --- |
|  | this.reactiveForm.valid |

**invalid**

invalid: boolean

A FormGroup is invalid when one of its controls has failed a validation check or the entire FormGroup is disabled.

|  |  |
| --- | --- |
|  | this.reactiveForm.invalid |

**pending**

pending: boolean

A FormGroup is pending when it is in the midst of conducting a validation check.

|  |  |
| --- | --- |
|  | this.reactiveForm.pending |

**disabled**

disabled: boolean

A FormGroup is disabled when all of its controls are disabled.

|  |  |
| --- | --- |
|  | this.reactiveForm.disabled |

**enabled**

enabled: boolean

A FormGroup is enabled as long one of its control is enabled.

|  |
| --- |
| this.reactiveForm.disabled |

**pristine**

pristine: boolean

A FormGroup is pristine if the user has not yet changed the value in the UI in any of the controls

|  |  |
| --- | --- |
|  | this.reactiveForm.pristine |

**dirty**

dirty: boolean

A FormGroup is dirty if the user has changed the value in the UI in any one of the control.

|  |  |
| --- | --- |
|  | this.reactiveForm.dirty |

**touched**

touched: boolean

True if the FomGroup is marked as touched. A FormGroup is marked as touched once the user has triggered a blur event on any one of the controls

|  |  |
| --- | --- |
|  | this.reactiveForm.touched |

**untouched**

untouched: boolean

True if the FormGroup has not been marked as touched. A FormGroup is untouched if the user has not yet triggered a blur event on any of its child controls

|  |
| --- |
| this.reactiveForm.untouched |

**Changing the Status**

We can also change the status of the FormGroup by using the following method.

**markAsTouched**

The FormGroup is marked as touched if anyone of its control is marked as touched. The control is marked as touched once the user has triggered a blur event on it.

markAsTouched(opts: { onlySelf?: boolean; } = {}): void

* onlySelf if true then only this control is marked. If false it will also mark all its direct ancestors also as touched. The default is false.

In the following example, the City is marked as touched. It will also mark both the address & reactiveFormGroup as touched.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsTouched() {      this.reactiveForm.get("address").get("city").markAsTouched();    } |

By Passing the onlySelf:true argument, you can ensure that only the city is marked as touched, while address & reactiveForm are not affected.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsTouched() {      this.reactiveForm.get("address").get("city").markAsTouched({onlySelf:true});    } |

The following code marks the address FormGroup as touched. while the child controls are not marked as touched. The parent FormGroup is marked as touched.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markAddressAsTouched() {      this.reactiveForm.get("address").markAsTouched();    } |

While onlySelf:true marks only the address group as touched, leaving the top-level FormGroup

|  |  |
| --- | --- |
| 5 | markAddressAsTouched() {      this.reactiveForm.get("address").markAsTouched({onlySelf:true});    } |

**markAllAsTouched**

Marks the control and all its descendant controls as touched.

markAllAsTouched(): void

The following example marks the address and all its controls i.e city, state & address as touched. The parent FormGroup stays as it is.

|  |  |
| --- | --- |
| 1 | markAllAddressTouched() {    this.reactiveForm.get("address").markAllAsTouched();  } |

**markAsUntouched**

Marks the control as untouched.

markAsUntouched(opts: { onlySelf?: boolean; } = {}): void

* onlySelf if true only this control is marked as untouched. When false or not supplied, mark all direct ancestors as untouched. The default is false.

The following code will mark the city as untouched. It will recalculate the untouched & touched status of the parent Group. If all the other controls are untouched then the parent FormGroup address is also marked as untouched.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsUnTouched() {    this.reactiveForm.get("address").get("city").markAsUntouched();  } |

By using the onlySelf:true you can ensure that only the city is marked as untouched, leaving the parent FormGroup as it is.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsUnTouched() {    this.reactiveForm.get("address").get("city").markAsUntouched({onlySelf:true});  } |

Similarly, you can mark the entire FormGroup as untouched. While this does not have any effect on the child controls, but it does recalculate the untouched status of the parent FormGroup. You can use the onlySelf:true ensure that it does not happen.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markAddressAsUnTouched() {    this.reactiveForm.get("address").markAsUntouched();  } |

**markAsDirty**

The FormGroup becomes dirty when any one of its control is marked as dirty. A control becomes dirty when the control’s value is changed through the UI. We can use the markAsDirty method to manipulate the dirty status.

markAsDirty(opts: { onlySelf?: boolean; } = {}): void

* onlySelf if true, only this control is marked as dirty else all the direct ancestors are marked as dirty. The default is false.

The Following code marks the entire form as dirty. It does not change the status of any of the child controls.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markFormAsDirty() {    this.reactiveForm.markAsDirty();  } |

The following code marks the City as dirty. It will also change the Dirty status of Parent FormGroup.

|  |  |
| --- | --- |
|  | markCityAsDirty() {    this.reactiveForm.get("address").get("city").markAsDirty();  } |

You can use the onlySelf:false to ensure that the parent FormGroup is not affected by our change.

|  |
| --- |
| markCityAsDirty() {      this.reactiveForm.get("address").get("city").markAsDirty({onlySelf:false});    } |

You can also make the entire FormGroup as dirty. It does not affect the child controls, but parent FormGroup is also marked as dirty. Unless you pass the {onlySelf:true} argument

|  |  |
| --- | --- |
|  | markAddressAsDirty() {    this.reactiveForm.get("address").markAsDirty({onlySelf:false});  } |

**markAsPristine**

The FormGroup becomes pristine when none of its controls values are changed via UI. The pristine is the opposite of dirty. We can use the markAsPrisitine method to manipulate the pristine status.

markAsPristine(opts: { onlySelf?: boolean; } = {}): void

* onlySelf if true, only this control is marked as pristine else all the direct ancestors are marked as pristine. The default is false.

The following code marks the Form as Pristine. It will also mark all the child controls as Pristine

|  |  |
| --- | --- |
|  | markFormAsPristine() {    this.reactiveForm.markAsPristine();  } |

The following code marks the city as Pristine. It will also calculate the Pristine status of the Parent FormGroup. If all the other controls are pristine then the parent FormGroup becomes pristine.

|  |  |
| --- | --- |
|  | markCityAsPristine() {      this.reactiveForm.get("address").get("city").markAsPristine({onlySelf:false});    } |

You can make use of the onlySelf:true to ensure that the pristine status of the parent group is not calculated.

**markAsPending**

Marks the control as pending. We usually use this when we running our validation checks. Pending means the status of the control cannot be determined at this time.

markAsPending(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

* onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. The default is false.
* emitEvent: When true or not supplied (the default), the statusChanges observable emits an event with the latest status the control is marked pending. When false, no events are emitted.

The following code marks the entire form as Pending. It does not change the status of child Controls.

|  |  |
| --- | --- |
|  | this.reactiveForm.markAsPending(); |

The following will mark the address FormGroup as Pending. It will also mark the Parent FormGroup as Pending also, which you can control using the onlySelf:true argument

|  |  |
| --- | --- |
|  | markAddressAsPendng() {    this.reactiveForm.get("address").markAsPending();  } |

This method also triggers the [statusChange](https://www.tektutorialshub.com/angular/statuschanges-in-angular-forms/) Event. You can make use of emitEvent:false argument, which will stop the statusChange event being triggered.

|  |  |
| --- | --- |
|  | markAddressAsPendng() {    this.reactiveForm.get("address").markAsPending({emitEvent:false});  } |

**disable**

Disables the control. This means the control is exempt from validation checks and excluded from the aggregate value of any parent. Its status is DISABLED.

disable(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

* onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. Default is false..
* emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is disabled. When false, no events are emitted.

The following code disables all the controls in the FormGroup.

|  |  |
| --- | --- |
|  | disableAll() {    this.reactiveForm.disable();  } |

If you disable all the controls individually, then the FormGroup is automatically disabled.

|  |  |
| --- | --- |
|  | disableAll() {    this.reactiveForm.get("firstname").disable();    this.reactiveForm.get("lastname").disable();    this.reactiveForm.get("email").disable();    this.reactiveForm.get("address").disable();  } |

**enable**

Enables control. This means the control is included in validation checks and the aggregate value of its parent. Its status recalculates based on its value and its validators.

enable(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

* onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. The default is false.
* emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is enabled. When false, no events are emitted.

The following command enables all the controls in the Group. Even the controls previously disabled are also enabled.

|  |  |
| --- | --- |
| 5 | enableAll() {     this.reactiveForm.enable();  } |

Enables only address FormGroup.

|  |  |
| --- | --- |
|  | enableAddress() {    this.reactiveForm.get("address").enable();  } |

Enable a Single Control

|  |  |
| --- | --- |
|  | enableFirstName() {    this.reactiveForm.get("firstname").enable();  } |

**Status Change Event**

**statusChanges**

statusChanges: Observable<any>

The statusChanges event is fired whenever the status of the form is calculated. We can subscribe to this event as shown below. We can subscribe it at the FormControl level or at the FormGroup level.

**Note that this event is fired whenever the status is calculated.**

In the example below, the first statusChanges is emitted, when the status of the top-level FormGroup is calculated. The second statusChange event is emitted, when the address FormGroup status is calculated.

|  |  |
| --- | --- |
|  | this.reactiveForm.statusChanges.subscribe(x => {    console.log(x);  })    this.reactiveForm.get("address").statusChanges.subscribe(x => {     console.log(x);  }) |

**Validation**

The validators can be added to FormControl, FormGroup or to the FormArray.

**updateValueAndValidity()**

The updateValueAndValidity forces the form to perform validation. When applied to the FormGroup, it will calculate the validity of all the child controls, including nested form groups & form arrays This is useful when you add/remove validators dynamically using setValidators, RemoveValidators etc

updateValueAndValidity(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

* onlySelf: When true, only update this control. When false or not supplied, update all direct ancestors. Default is false..
* emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is updated. When false, no events are emitted.

|  |  |
| --- | --- |
|  | this.reactiveForm.updateValueAndValidity();  this.reactiveForm.get("address").updateValueAndValidity(); |

**setValidators() / setAsyncValidators()**

Programmatically adds the sync or async validators. This method will remove all the previously added sync or async validators.

setValidators(newValidator: ValidatorFn | ValidatorFn[]): void  
setAsyncValidators(newValidator: AsyncValidatorFn | AsyncValidatorFn[]): void

|  |  |
| --- | --- |
|  | setValidator() {    this.reactiveForm.get("address").setValidators([addressValidator]);    this.reactiveForm.get("address").updateValueAndValidity();  }      export const addressValidator = (control: AbstractControl): {[key: string]: boolean} => {    const city = control.get('city').value;    const state = control.get('state').value;    console.log(control.value);    if (city=="" && state=="") {      return { address:false };    }    return null;  }; |

**clearValidators() / clearAsyncValidators()**

clearValidators(): void  
clearAsyncValidators(): void

clearValidators & clearAsyncValidators clears all validators.

|  |
| --- |
| //reactive forms  clearValidation() {     this.reactiveForm.get("address").clearValidators();     this.reactiveForm.get("address").updateValueAndValidity();  } |

**errors()**

errors: ValidationErrors | null

An object containing any errors generated by failing validation, or null if there are no errors.

|  |
| --- |
| getErrors() {      const controlErrors: ValidationErrors = this.reactiveForm.errors;    if (controlErrors) {      Object.keys(controlErrors).forEach(keyError => {        console.log("firtname "+ ' '+keyError);      });    }  } |

**setErrors()**

setErrors(errors: ValidationErrors, opts: { emitEvent?: boolean; } = {}): void

|  |  |
| --- | --- |
|  | setErrors() {    this.reactiveForm.setErrors( {customerror:'custom error'});  } |

**getError()**

getError(errorCode: string, path?: string | (string | number)[]): any

Reports error data for the control with the given path.

|  |  |
| --- | --- |
|  | this.reactiveForm.getError("firstname")    this.reactiveForm.getError("address.pincode");  this.reactiveForm.getError(["address","pincode"]); |

**hasError**

hasError(errorCode: string, path?: string | (string | number)[]): boolean

Reports whether the control with the given path has the error specified.

|  |
| --- |
| this.reactiveForm.hasError("firstname")    //  this.reactiveForm.hasError("address.pincode");  this.reactiveForm.hasError(["address","pincode"]); |

**Reset**

abstract reset(value?: any, options?: Object): void

Resets the control. We can also pass the default value.

|  |  |
| --- | --- |
| 1  2  3  4  5 | this.reactiveForm.get("firstname").reset('');  this.reactiveForm.get("firstname").reset('test'); |

**Summary**

In this tutorial, we learned what is FormControl is and looked at the various methods & properties that are available.

Example-1 checkbox example with reactive forms

Step 1: Import FormsModule

If you want to create form in angular app then you need to import FormsModule from @angular/forms library. so let's add following code to app.module.ts file.

**src/app/app.module.ts**

**import** { BrowserModule } **from** '@angular/platform-browser';

**import** { NgModule } **from** '@angular/core';

**import** { AppComponent } **from** './app.component';

**import** { FormsModule, ReactiveFormsModule } **from** '@angular/forms';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule

],

providers: [],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }

Step 2: Form with ngModel

In this step, we will write code of html form with ngModel. so add following code to app.component.html file.

I used bootstrap class on this form. if you want to add than then follow this link too: [Install Boorstrap 4 to Angular 10](https://www.itsolutionstuff.com/post/angular-10-bootstrap-modal-exampleexample.html).

**src/app/app.component.html**

**<div** **class**="container"**>**

**<h1>**Angular Radio Button Example - ItSolutionStuff.com**</h1>**

**<form** [**formGroup**]="form" (**ngSubmit**)="submit()"**>**

**<div** **class**="form-group"**>**

**<label** **for**="gender"**>**Gender:**</label>**

**<div>**

**<input** **id**="male" **type**="radio" **value**="male" **name**="gender" **formControlName="gender">**

**<label** **for**="male"**>**Male**</label>**

**</div>**

**<div>**

**<input** **id**="female" **type**="radio" **value**="female" **name**="gender" **formControlName**="gender"**>**

**<label** **for**="female"**>**Female**</label>**

**</div>**

**<div** \***ngIf**="f.gender.touched && f.gender.invalid" **class**="alert alert-danger"**>**

**<div** \***ngIf**="f.gender.errors.required"**>**Name is required.**</div>**

**</div>**

**</div>**

**<button** **class**="btn btn-primary" **type**="submit" [**disabled**]="!form.valid"**>**Submit**</button>**

**</form>**

**</div>**

Read Also: [Angular 11/10 Create Custom Pipe Example](https://www.itsolutionstuff.com/post/angular-10-create-custom-pipe-exampleexample.html)

Step 3: updated Ts File

In ts file. we will write submit() and get all input fields values. so let's add following code to app.component.ts file.

**src/app/app.component.ts**

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

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

**export** **class** AppComponent {

form = **new** FormGroup({

gender: **new** FormControl('', Validators.required)

});

**get** f(){

**return** **this**.form.controls;

}

submit(){

console.log(**this**.form.value);

}

}

Now you can run your application using following command:

ng serve

Example 2: Get Checked Radio Button on Change Event

Step 1: Import FormsModule

If you want to create form in angular app then you need to import FormsModule from @angular/forms library. so let's add following code to app.module.ts file.

**src/app/app.module.ts**

**import** { BrowserModule } **from** '@angular/platform-browser';

**import** { NgModule } **from** '@angular/core';

**import** { AppComponent } **from** './app.component';

**import** { FormsModule, ReactiveFormsModule } **from** '@angular/forms';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule

],

providers: [],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }

Step 2: Form with ngModel

In this step, we will write code of html form with ngModel. so add following code to app.component.html file.

I used bootstrap class on this form. if you want to add than then follow this link too: [Install Boorstrap 4 to Angular 10](https://www.itsolutionstuff.com/post/angular-10-bootstrap-modal-exampleexample.html).

**src/app/app.component.html**

**<div** **class**="container"**>**

**<h1>**Angular Radio Button Example - ItSolutionStuff.com**</h1>**

**<form** [**formGroup**]="form" (**ngSubmit**)="submit()"**>**

**<div** **class**="form-group"**>**

**<label** **for**="gender"**>**Gender:**</label>**

**<div>**

**<input** **id**="male" **type**="radio" **value**="male" **name**="gender" **formControlName**="gender" (**change**)="changeGender($event)"**>**

**<label** **for**="male"**>**Male**</label>**

**</div>**

**<div>**

**<input** **id**="female" **type**="radio" **value**="female" **name**="gender" **formControlName**="gender" (**change**)="changeGender($event)"**>**

**<label** **for**="female"**>**Female**</label>**

**</div>**

**<div** \***ngIf**="f.gender.touched && f.gender.invalid" **class**="alert alert-danger"**>**

**<div** \***ngIf**="f.gender.errors.required"**>**Name is required.**</div>**

**</div>**

**</div>**

**<button** **class**="btn btn-primary" **type**="submit" [**disabled**]="!form.valid"**>**Submit**</button>**

**</form>**

**</div>**

Read Also: [Angular 11/10 Create Custom Pipe Example](https://www.itsolutionstuff.com/post/angular-10-create-custom-pipe-exampleexample.html)

Step 3: updated Ts File

In ts file. we will write submit() and get all input fields values. so let's add following code to app.component.ts file.

**src/app/app.component.ts**

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

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

**export** **class** AppComponent {

form = **new** FormGroup({

gender: **new** FormControl('', Validators.required)

});

**get** f(){

**return** **this**.form.controls;

}

submit(){

console.log(**this**.form.value);

}

changeGender(e) {

console.log(e.target.value);

}

}

Now you can run your application using following command:

# Example-3

# Angular 11/10 Select Dropdown Example Tutorial

You need to follow bellow step to add form validation in angular 10.

**Example 1: Get Selected DropDown Value on Form Submit**

**Step 1: Import FormsModule**

If you want to create form in angular app then you need to import FormsModule from @angular/forms library. so let's add following code to app.module.ts file.

**src/app/app.module.ts**

**import** { BrowserModule } **from** '@angular/platform-browser';

**import** { NgModule } **from** '@angular/core';

**import** { AppComponent } **from** './app.component';

**import** { FormsModule, ReactiveFormsModule } **from** '@angular/forms';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule

],

providers: [],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }

**Step 2: Form with ngModel**

In this step, we will write code of html form with ngModel. so add following code to app.component.html file.

I used bootstrap class on this form. if you want to add than then follow this link too: [Install Boorstrap 4 to Angular 10](https://www.itsolutionstuff.com/post/how-to-use-bootstrap-4-in-angular-10example.html).

**src/app/app.component.html**

**<div** **class**="container"**>**

**<h1>**Angular Select Dropdown Example - ItSolutionStuff.com**</h1>**

**<form** [**formGroup**]="form" (**ngSubmit**)="submit()"**>**

**<div** **class**="form-group"**>**

**<label** **for**="website"**>**Website:**</label>**

**<select** **formControlName**="website" **class**="form-control"**>**

**<option** **disabled>**Select Website**</option>**

**<option>**Choose Website**</option>**

**<option** \***ngFor**="let web of websiteList"**>**{{web}}**</option>**

**</select>**

**<div** \***ngIf**="f.website.touched && f.website.invalid" **class**="alert alert-danger"**>**

**<div** \***ngIf**="f.website.errors.required"**>**Name is required.**</div>**

**</div>**

**</div>**

**<button** **class**="btn btn-primary" **type**="submit" [**disabled**]="!form.valid"**>**Submit**</button>**

**</form>**

**</div>**

Read Also: [10 Digit Mobile Number Validation in Angular](https://www.itsolutionstuff.com/post/10-digit-mobile-number-validation-in-angularexample.html)

**Step 3: updated Ts File**

In ts file. we will write submit() and get all input fields values. so let's add following code to app.component.ts file.

**src/app/app.component.ts**

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

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

**export** **class** AppComponent {

websiteList: any = ['ItSolutionStuff.com', 'HDTuto.com', 'Nicesnippets.com']

form = **new** FormGroup({

website: **new** FormControl('', Validators.required)

});

**get** f(){

**return** **this**.form.controls;

}

submit(){

console.log(**this**.form.value);

}

}

Now you can run your application using following command:

ng serve

**Preview:**

**Example 2: Get Selected DropDown Value on Change Event**

**Step 1: Import FormsModule**

If you want to create form in angular app then you need to import FormsModule from @angular/forms library. so let's add following code to app.module.ts file.

**src/app/app.module.ts**

**import** { BrowserModule } **from** '@angular/platform-browser';

**import** { NgModule } **from** '@angular/core';

**import** { AppComponent } **from** './app.component';

**import** { FormsModule, ReactiveFormsModule } **from** '@angular/forms';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule

],

providers: [],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }

**Step 2: Form with ngModel**

In this step, we will write code of html form with ngModel. so add following code to app.component.html file.

I used bootstrap class on this form. if you want to add than then follow this link too: [Install Boorstrap 4 to Angular 10](https://www.itsolutionstuff.com/post/how-to-use-bootstrap-4-in-angular-10example.html).

**src/app/app.component.html**

**<div** **class**="container"**>**

**<h1>**Angular Select Dropdown Example - ItSolutionStuff.com**</h1>**

**<form** [**formGroup**]="form" (**ngSubmit**)="submit()"**>**

**<div** **class**="form-group"**>**

**<label** **for**="website"**>**Website:**</label>**

**<select** **formControlName**="website" **class**="form-control" (**change**)="changeWebsite($event)"**>**

**<option** **disabled>**Select Website**</option>**

**<option>**Choose Website**</option>**

**<option** \***ngFor**="let web of websiteList"**>**{{web}}**</option>**

**</select>**

**<div** \***ngIf**="f.website.touched && f.website.invalid" **class**="alert alert-danger"**>**

**<div** \***ngIf**="f.website.errors.required"**>**Name is required.**</div>**

**</div>**

**</div>**

**<button** **class**="btn btn-primary" **type**="submit" [**disabled**]="!form.valid"**>**Submit**</button>**

**</form>**

**</div>**

Read Also: [10 Digit Mobile Number Validation in Angular](https://www.itsolutionstuff.com/post/10-digit-mobile-number-validation-in-angularexample.html)

**Step 3: updated Ts File**

In ts file. we will write submit() and get all input fields values. so let's add following code to app.component.ts file.

**src/app/app.component.ts**

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

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

**export** **class** AppComponent {

websiteList: any = ['ItSolutionStuff.com', 'HDTuto.com', 'Nicesnippets.com']

form = **new** FormGroup({

website: **new** FormControl('', Validators.required)

});

**get** f(){

**return** **this**.form.controls;

}

submit(){

console.log(**this**.form.value);

}

changeWebsite(e) {

console.log(e.target.value);

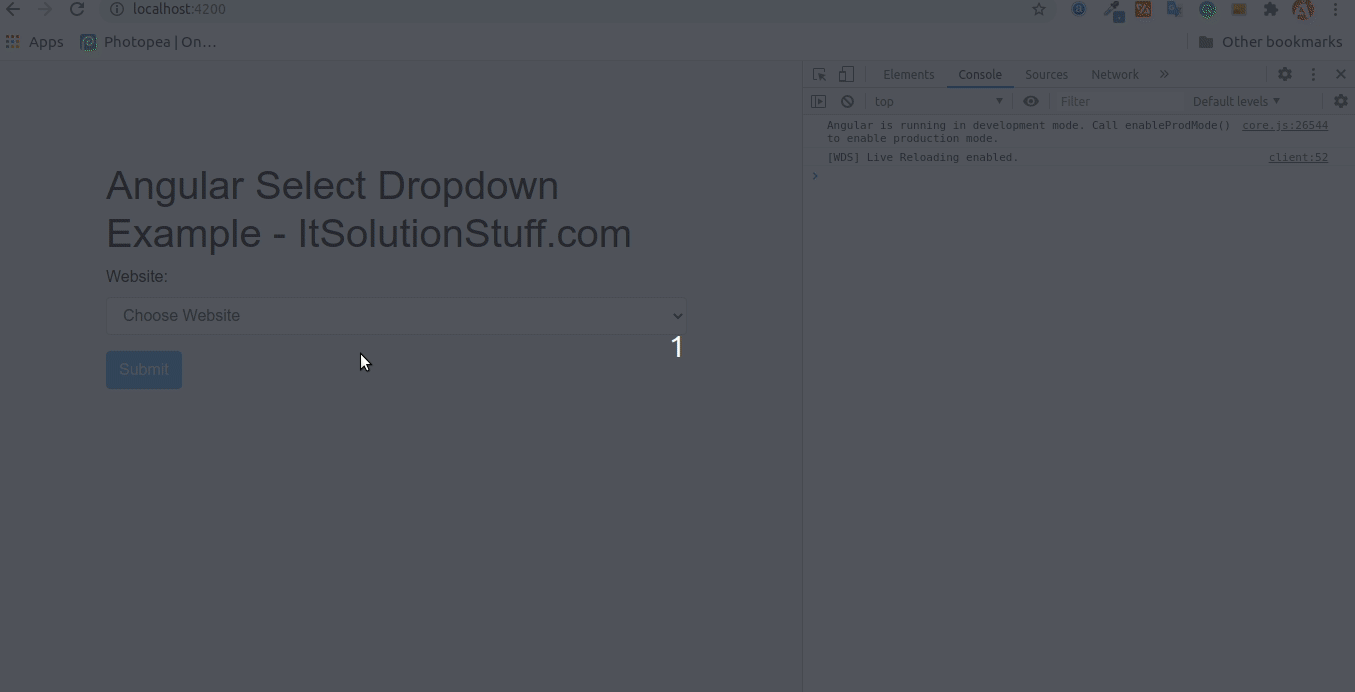
}

}

Now you can run your application using following command:

ng serve

**Preview:**



**Reserved • www.itsolutionstuff.com**

**Reactive Form Validation In Angular 11**

* Step 1 – Create New Angular App
* Step 2 – Import Form Module
* Step 3 – Add Code on View File
* Step 4 – Use Component ts File
* Step 5 – Start Angular App

**Step 1 – Create New Angular App**

First of all, open your terminal and execute the following command on it to install angular app:

ng new my-new-app

**Step 2 – Import Module**

Then, Open app.module.ts file and import HttpClientModule, FormsModule and ReactiveFormsModule to app.module.ts file like following:

|  |
| --- |
| import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule, ReactiveFormsModule } from '@angular/forms';    import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      FormsModule,      ReactiveFormsModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

**Step 3 – Add Code on View File**

In this step, create simple reactive form with input file element. So, visit src/app/app.component.html and update the following code into it:

|  |  |
| --- | --- |
|  | <h1>Angular 11 Reactive Forms Validation Example - Tutsmake.com</h1>    <form [formGroup]="form" (ngSubmit)="submit()">        <div class="form-group">          <label for="name">Name</label>          <input              formControlName="name"              id="name"              type="text"              class="form-control">          <div \*ngIf="f.name.touched && f.name.invalid" class="alert alert-danger">              <div \*ngIf="f.name.errors.required">Name is required.</div>              <div \*ngIf="f.name.errors.minlength">Name should be 3 character.</div>          </div>      </div>        <div class="form-group">          <label for="email">Email</label>          <input              formControlName="email"              id="email"              type="text"              class="form-control">          <div \*ngIf="f.email.touched && f.email.invalid" class="alert alert-danger">              <div \*ngIf="f.email.errors.required">Email is required.</div>              <div \*ngIf="f.email.errors.email">Please, enter valid email address.</div>          </div>      </div>        <div class="form-group">          <label for="body">Body</label>          <textarea              formControlName="body"              id="body"              type="text"              class="form-control">          </textarea>          <div \*ngIf="f.body.touched && f.body.invalid" class="alert alert-danger">              <div \*ngIf="f.body.errors.required">Body is required.</div>          </div>      </div>        <button class="btn btn-primary" type="submit">Submit</button>  </form> |

**Step 4 – Use Component ts File**

In this step, visit the **src/app directory and open app.component.ts**. Then add the following code like formGroup and formControl element on component.ts file:

|  |  |
| --- | --- |
|  | import { Component } from '@angular/core';  import { FormGroup, FormControl, Validators} from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {      form = new FormGroup({      name: new FormControl('', [Validators.required, Validators.minLength(3)]),      email: new FormControl('', [Validators.required, Validators.email]),      body: new FormControl('', Validators.required)    });      get f(){      return this.form.controls;    }      submit(){      console.log(this.form.value);    }    } |

**Step 5 – Start Angular App**

In this step, execute the following command on terminal to start angular app:

Example-5

**Angular 12 Reactive Forms Validation Example**

You have to assimilate the following guide to create and validate reactive forms in angular 12 application.

* Create Angular Project
* Install Bootstrap Package
* Import ReactiveFormsModule in App Module
* Create Form Component
* Update the Main App Component
* Test Angular Form Validation App

**Create Angular Project**

Before starting this tutorial, first, make sure you have installed the Angular CLI globally on your device:

npm install -g @angular/cli

Bash

After that, you need to invoke the installation of a new angular project:

ng new ng-form-validaiton

Bash

Make sure to move to the project root:

cd ng-form-validaiton

Bash

**Install Bootstrap Package**

Make sure to install Bootstrap library in angular for creating immaculate forms; consequently, it helps you create sleek, intuitive and robust front-end in exorbitantly low time.

npm i bootstrap

TypeScript

COPY

Add Bootstrap library CSS path in **angular.json** file:

"styles": [

"node\_modules/bootstrap/dist/css/bootstrap.min.css",

"src/styles.css"

]

TypeScript

COPY

**Add Reactive Forms Module**

For using the reactive form controls, must register ReactiveFormsModule likewise FormsModule from the @angular/forms package and insert inside the NgModule’s imports array.

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { ReactiveFormsModule, FormsModule } from '@angular/forms';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

ReactiveFormsModule,

FormsModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

TypeScript

COPY

**Create Form Component**

In this step, you have to generate a new component using the following command:

**ng g c reactive-form**

Bash

Open **reactive-form.component.ts** file, in the same way update the below code:

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

import { FormGroup, FormBuilder, Validators } from "@angular/forms";

@Component({

selector: 'app-reactive-form',

templateUrl: './reactive-form.component.html',

styleUrls: ['./reactive-form.component.css']

})

export class ReactiveFormComponent implements OnInit {

userForm: FormGroup;

constructor(public formBuilder: FormBuilder) { }

ngOnInit(): void {

this.userForm = this.formBuilder.group({

name: ['', [Validators.required, Validators.minLength(4)]],

email: ['', [Validators.required]],

phone: ['', [Validators.required]],

message: ['', [Validators.required]]

})

}

get getControl(){

return this.userForm.controls;

}

onSubmit(){

console.log(this.userForm);

}

}

TypeScript

To create and validate form import FormGroup, FormBuilder and Validators modules from “@angular/forms” package.

Insert FormBuilder in the constructor, also evoke the form object using FormGroup class.

Inside the ngOnInit lifecycle hook, declare the form inside the group method. Even more add a name, email, phone and message form controls respectively and gradually.

To implement validation in angular form inputs inject the Validators class, It offers a couple of ready-made validators which can be addressed by form controls instances.

Lastly, to shorten up the process of accessing the form control instances use the getter method.

Open **reactive-form.component.html** file, on top of that insert the below code:

<h2 class="mb-5 text-center">Angular 12 Form Validation Example</h2>

<form [formGroup]="userForm" (ngSubmit)="onSubmit()" novalidate>

<div class="form-group">

<label>Name</label>

<input type="text" class="form-control" formControlName="name">

<div \*ngIf="getControl.name.touched && getControl.name.invalid" class="text-danger">

<div \*ngIf="getControl.name.errors.required">Name is required.</div>

<div \*ngIf="getControl.name.errors?.minlength">Minimum 4 character are required.</div>

</div>

</div>

<div class="form-group">

<label>Email</label>

<input type="text" class="form-control" formControlName="email">

<div \*ngIf="getControl.email.touched && getControl.email.invalid" class="text-danger">

<div \*ngIf="getControl.email.errors.required">Email is required.</div>

<div \*ngIf="getControl.email.errors.email">Email seems invalid.</div>

</div>

</div>

<div class="form-group">

<label>Phone</label>

<input type="text" class="form-control" formControlName="phone">

<div \*ngIf="getControl.phone.touched && getControl.phone.invalid" class="text-danger">

<div \*ngIf="getControl.phone.errors.required">Phone is required.</div>

</div>

</div>

<div class="form-group">

<label>Message</label>

<textarea type="text" class="form-control" formControlName="message"></textarea>

<div \*ngIf="getControl.message.touched && getControl.message.invalid" class="text-danger">

<div \*ngIf="getControl.message.errors.required">Message is required.</div>

</div>

</div>

<button class="btn btn-danger btn-block" type="submit">Submit</button>

</form>

Create form object using formGoup directive, additionally define the submit event with novalidate property.

Make the form inputs ready by adding formControlName property, and the form control will contain the form input value.

As you can see, we access the form errors, and touched props and added the form validation for required, minimum characters and touched.

**Update the Main App Component**

Copy Component selector from ReactiveFormComponent file and declare within the app component file, so open **app.component.html** file, in the second place add the following code:

<div class="container mt-5" style="max-width: 550px;">

<app-reactive-form></app-reactive-form>

</div>

Markup

COPY

**Test Angular Form Validation App**

Eventually, we have completed all the steps to build the angular reactive form; now, we have to test the application.

ng serve

Bash

COPY

Open browser, enter the below url:

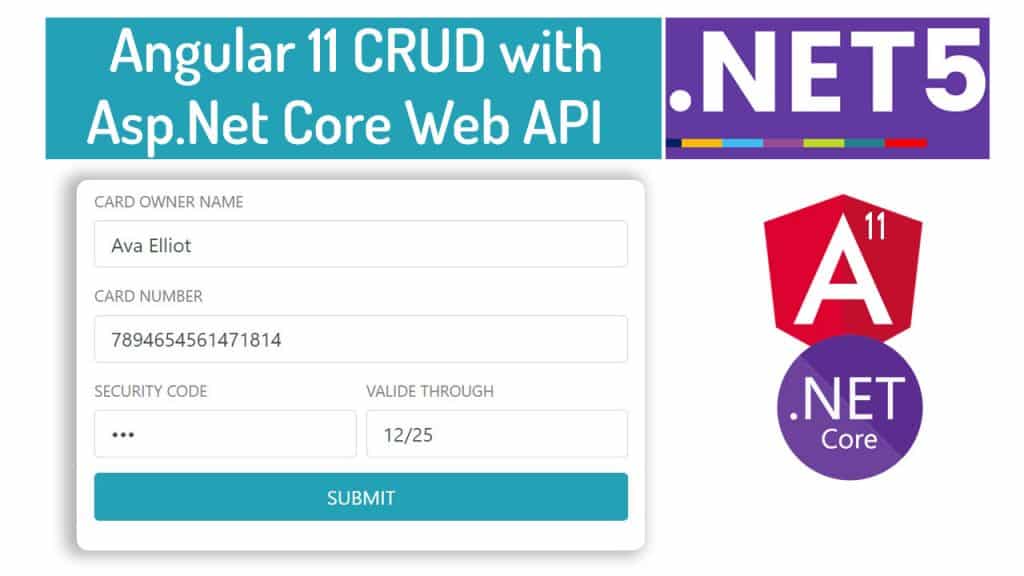
http://localhost:4200

**Conclusion**

Ultimately, this tutorial is over. Using this profound guide, now you can create an angular form with reactive forms and validate using reactive forms validator API. I reckon you would love this tutorial and propel forward to other developer friends.

Example-7

# Angular 11 CRUD with ASP.NET Core Web API



In this article, we’ll implement **Asp.Net Core 5.0 Web API CRUD Operations with Angular 11**. To demonstrate the topic, we’ll build a project from scratch with payment details like credit/ debit card.

GitHub repository: <https://bit.ly/3qyiusO>

Sub-topics discussed.

* ASP.NET Core Web API
  + Create .NET Core Web API
  + Setup Database with EF Core
  + API Controller for CRUD Web Methods
  + Enable CORS for Angular App
* Angular Client Side
  + Create Angular 11 Project
  + Consume ASP.NET Core API From Angular
  + Form Design and Validation
  + Insert/ Create Record by Form Submission
  + Retrieve and Display Inserted Records
  + Update and Delete Operation

**Table of Contents**

[Create ASP.NET Core Web API](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Create_ASPNET_Core_Web_API)

[Setup Database](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Setup_Database)

[Create API Controller for CRUD Operations](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Create_API_Controller_for_CRUD_Operations)

[Create Angular App](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Create_Angular_App)

[How to Consume .Net Core API from Angular](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#How_to_Consume_Net_Core_API_from_Angular)

[Form Design and Validation](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Form_Design_and_Validation)

[Insert a new Record](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Insert_a_new_Record)

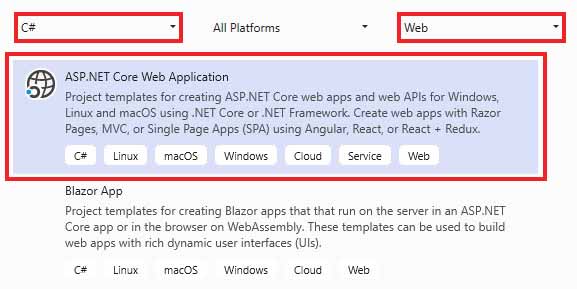
[Retrieve and Display all Inserted Records](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Retrieve_and_Display_all_Inserted_Records)

[Update and Delete Operation](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Update_and_Delete_Operation)

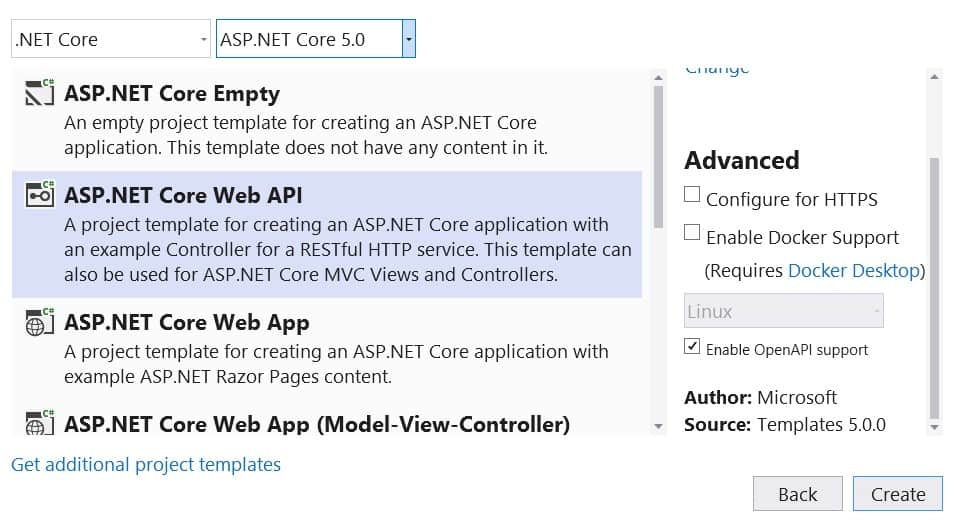
[Video Tutorial](https://www.codaffection.com/asp-net-core-article/angular-crud-with-asp-net-core-web-api/#Video_Tutorial)

## Create ASP.NET Core Web API

In Visual Studio 2019, From the new project window, select Asp.Net Core Web Application.

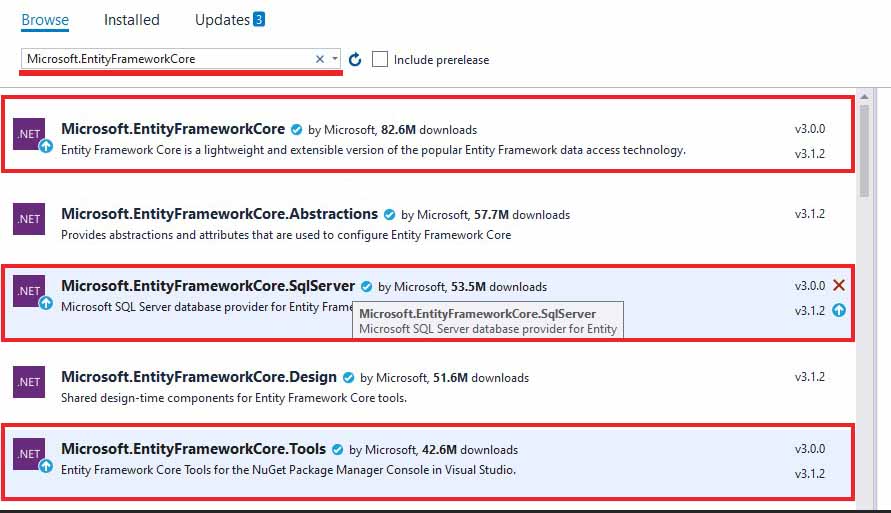


Once you provide the project name and location. A new window will be opened as follows, Select API. The above steps will create a brand new ASP.NET Core Web API project.



## Setup Database

Let’s create a Database for this project. Inside this project, we’ll be using Entity Framework Core to create and interact with the database. So first of all we’ve to install corresponding NuGet packages. Right-click on the project name from Solution Explorer, click on Manage NuGet Packages, from Browse Tab, install the following 3 packages with same version as that of Asp.Net Core.



Now, let’s define DB model class file –PaymentDetail.cs in a new folder Models.

public class PaymentDetail

{

[Key]

public int PaymentDetailId { get; set; }

[Required]

[Column(TypeName = "nvarchar(100)")]

public string CardOwnerName { get; set; }

[Required]

[Column(TypeName = "varchar(16)")]

public string CardNumber { get; set; }

[Required]

[Column(TypeName = "varchar(5)")]

public string ExpirationDate { get; set; }

[Required]

[Column(TypeName = "varchar(3)")]

public string SecurityCode { get; set; }

}

C#

Copy

Now let’s define DbContext class file- /Models/PaymentDetailContext.cs.

public class PaymentDetailContext : DbContext

{

public PaymentDetailContext(DbContextOptions<PaymentDetailContext> options):base(options)

{ }

public DbSet<PaymentDetail> PaymentDetails { get; set; }

}

C#

Copy

DbContext class- PaymentDetailContext decides what should be added to actual physical database during DB Migration. So we have added DbSet property for PaymentDetail Model class, after migration PaymentDetails table will be created in SQL Server Database.

Into this model class constructor parameter- options , we have to pass which DbProvider (SQL Server, MySQL, PostgreSQL, etc) to use and corresponding DB connection string also. For that, we’ll be using dependency injection in ASP.NET Core with Startup.cs file as follows.

public void ConfigureServices(IServiceCollection services)

{

...

services.AddDbContext<PaymentDetailContext>(optionns =>

optionns.UseSqlServer(Configuration.GetConnectionString("DevConnection")));

}

C#

Copy

Here we’ve used dependency injection for DbContext class, through which SQL Server is set as a DbProvider with a connection string, Now save the connection string in appsettings.json file using DevConnection key as follows.

{

....

"ConnectionStrings": {

"DevConnection": "Server=(local)\\sqlexpress;Database=PaymentDetailDB;Trusted\_Connection=True;MultipleActiveResultSets=True;"

}

}

JSON

Copy

Now let’s do the migration. Select project from solution explorer, then go to **Tools > NuGet Package Manager > Package Manager Console**. Then execute following commands one by one.

Add-Migration "InitialCreate"

Update-Database

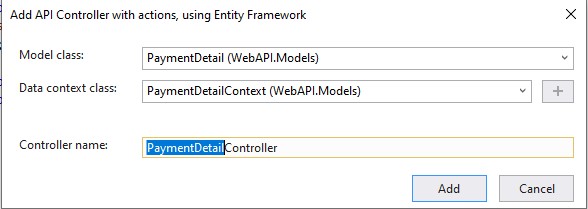
none

Copy

After successful migration, as per the connection string, a new database – PaymentDetailDB will be created with PaymentDetails table. Also, there will be a new Migrations folder created with corresponding C# files.

## Create API Controller for CRUD Operations

To create a new API controller, right-click on Controllers folder **Add > Controller**, Select API Controller with actions, using Entity Framework.



With the help of scaffolding mechanism, newly created PaymentDetailController will look like this.

[Route("api/[controller]")]

[ApiController]

public class PaymentDetailController : ControllerBase

{

private readonly PaymentDetailContext \_context;

public PaymentDetailController(PaymentDetailContext context)

{

\_context = context;

}

// GET: api/PaymentDetail

[HttpGet]

public async Task<ActionResult<IEnumerable<PaymentDetail>>> GetPaymentDetails()

{ ... }

// GET: api/PaymentDetail/5

[HttpGet("{id}")]

public async Task<ActionResult<PaymentDetail>> GetPaymentDetail(int id)

{ ... }

// PUT: api/PaymentDetail/5

[HttpPut("{id}")]

public async Task<IActionResult> PutPaymentDetail(int id, PaymentDetail paymentDetail)

{ ... }

// POST: api/PaymentDetail

[HttpPost]

public async Task<ActionResult<PaymentDetail>> PostPaymentDetail(PaymentDetail paymentDetail)

{ ... }

// DELETE: api/PaymentDetail/5

[HttpDelete("{id}")]

public async Task<ActionResult<PaymentDetail>> DeletePaymentDetail(int id)

{ ... }

private bool PaymentDetailExists(int id)

{ ... }

}

C#

Copy

It contains web methods POST, GET, PUT and DELETE for Create, Retrieve, Update and Delete operations respectively. As a constructor parameter we’ve context of the type PaymentDetailContext . the instance/value for this parameter will be passed from dependency injection from StartUp class.

For this project, we don’t have to change anything in web methods and you can test any of these CRUD operations using software like [postman](https://www.postman.com/) or you use open api support with the swagger interface.

## Create Angular App

Now let’s create front-end client-side app in Angular 11. For that execute following Angular-CLI commands one by one.

ng new app\_name

# after project creation.

# navigate inside project folder

cd app\_name

# open in vs code

code .

# from vs code terminal

# command to open the app in default web browser

ng serve --o

none

Copy

Before moving forward, let’s look at the structure of the app that we want to build.

● src

+---● app

| +--● payment-details

| | |--payment-details.component.ts|.html

| | |

| | +--● payment-detail-form

| | |--payment-detail-form.component.ts|.html

| |

| +--● shared

| | |--payment-detail.service.ts

| | |--payment-detail.model.ts

| |

| |--app.module.ts

|

|--index.html (cdn path for bootstrap & fa icons)

None

Copy

We’ve two components, list of records will be shown in payment-details component, it has a child component payment-detail-form, inside that, a form for insert and update operation can be designed.

To create these 2 components, you can execute the following commands.

ng g c payment-details -s --skipTests

ng g c payment-details/payment-detail-form -s --skipTests

none

Copy

Options used  
inlinestyle | -s : skip seperate component specific stylesheet  
---skipTests : skip test files with extension .spec.ts

Now let’s replace the default component file- app.component.html as follows.

<div class="container">

<app-payment-details></app-payment-details>

</div>

HTML

Copy

To show list of records and it’s form side by side, update payment-details.component.html as bellow.

<div class="jumbotron py-3">

<h1 class="display-4 text-center">Payment Detail Register</h1>

</div>

<div class="row">

<div class="col-md-5">

<app-payment-detail-form></app-payment-detail-form>

</div>

<div class="col-md-7">

<div>table with list of records from the table</div>

</div>

</div>

HTML

Copy

For this app development, we’ll be using Bootstrap and Font Awesome Icons. so let’s add their stylesheet reference in index.html.

<!-- Add following stylesheets to <head> tag -->

<link rel="preload" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" data-rocket-async="style" as="style" onload="this.onload=null;this.rel='stylesheet'" integrity="sha384-JcKb8q3iqJ61gNV9KGb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN5t9UJ0Z" crossorigin="anonymous">

<link rel="preload" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.14.0/css/all.min.css" data-rocket-async="style" as="style" onload="this.onload=null;this.rel='stylesheet'" integrity="sha512-1PKOgIY59xJ8Co8+NE6FZ+LOAZKjy+KY8iq0G4B3CyeY6wYHN3yt9PW0XpSriVlkMXe40PTKnXrLnZ9+fkDaog==" crossorigin="anonymous" />

HTML

Copy

I’ve few custom CSS to add in global style sheet – styles.css.

.form-group label {

color: grey;

}

.form-group input {

font-weight: 500;

}

.form-group input::placeholder {

color: #c0bdbd;

font-weight: 300;

}

thead th{

font-weight: 400;

}

table tr:hover {

background-color: #fffbf2;

cursor: pointer;

}

/\*for invalid form controls\*/

input.invalid {

border-color: red;

}

CSS

Copy

## How to Consume .Net Core API from Angular

First of all, let’s create a model class for payment details – shared/payment-detail.model.ts. You could manually create the file with the following CLI command.

ng g class shared/payment-detail --type=model --skipTests

# there is no seperat command for model

# hence we use class creation command with type option

# type is used to name the file like '.model.ts'

none

Copy

Update the model class with corresponding properties similar to .Net Core API model properties.

export class PaymentDetail {

paymentDetailId: number=0;

cardOwnerName: string='';

cardNumber: string='';

expirationDate: string='';

securityCode: string='';

}

C#

Copy

Now let’s create a service class to interact with ASP.NET Core Web API- shared/payment-detail.service.ts. Here is the CLI command to create the service class.

ng g s shared/payment-detail --skipTests

none

Copy

Update the service class as below.

import { PaymentDetail } from './payment-detail.model';

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

import { HttpClient } from "@angular/common/http";

@Injectable({

providedIn: 'root'

})

export class PaymentDetailService {

formData: PaymentDetail= new PaymentDetail();

readonly baseURL = 'http://localhost:61236/api/PaymentDetail';

list : PaymentDetail[];

constructor(private http: HttpClient) { }

postPaymentDetail() {

return this.http.post(this.baseURL, this.formData);

}

putPaymentDetail() {

return this.http.put(`${this.baseURL}/${this.formData.paymentDetailId}`, this.formData);

}

deletePaymentDetail(id: number) {

return this.http.delete(`${this.baseURL}/${id}`);

}

refreshList() {

this.http.get(this.baseURL)

.toPromise()

.then(res =>this.list = res as PaymentDetail[]);

}

}

JavaScript

Copy

formData property can be used for designing the form for CRUD Operations, list array is used to store all of the records from the API. baseURL contains the base URL for the Web API controller. Now let’s run the API from Visual Studio – **Debug > Start Debugging(F5)**. HttpClient is used to make HTTP request to the server. Along with methods for CRUD operations, we’ve refreshList function to populate existing records into list property.

To use HttpClient , we also need to import HttpClientModule . So update app/app.module.ts as follows.

...

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

@NgModule({

...

imports: [HttpClientModule, ...],

...

})

TypeScript

Copy

## Form Design and Validation

Now let’s design the form in payment-detail-form component.

import { PaymentDetailService } from './../../shared/payment-detail.service';

import { PaymentDetail } from 'src/app/shared/payment-detail.model';

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

import { NgForm } from '@angular/forms';

@Component({

selector: 'app-payment-detail-form',

templateUrl: './payment-detail-form.component.html',

styles: []

})

export class PaymentDetailFormComponent implements OnInit {

constructor(public service: PaymentDetailService) { }

ngOnInit():void {

}

resetForm(form: NgForm) {

form.form.reset();

this.service.formData = new PaymentDetail();

}

}

TypeScript

Copy

Since we’ve the service injected here. we can access it’s property formData in component HTML file for designing the form. resetForm function will reset the form to its initiale state.

As a first step towards designing the form, we’ve to import FormsModule in app/app.module.ts.

...

import { FormsModule } from '@angular/forms';

@NgModule({

...

imports: [FormsModule, ...],

...

})

TypeScript

Copy

Now let’s design the form. so update payment-detail-form.component.html as shown below.

<form novalidate autocomplete="off" #form="ngForm" (submit)="onSubmit(form)">

<input type="hidden" name="paymentDetailId" [value]="service.formData.paymentDetailId" />

<div class="form-group">

<label>CARD OWNER NAME</label>

<input class="form-control form-control-lg" placeholder="Full Name" name="cardOwnerName"

#cardOwnerName="ngModel" [(ngModel)]="service.formData.cardOwnerName"

required [class.invalid]="cardOwnerName.invalid &#038;&#038; cardOwnerName.touched">

</div>

<div class="form-group">

<label>CARD NUMBER</label>

<input class="form-control form-control-lg" placeholder="16 Digit Card Number" name="cardNumber"

#cardNumber="ngModel" [(ngModel)]="service.formData.cardNumber"

required maxlength="16" minlength="16" [class.invalid]="cardNumber.invalid &#038;&#038; cardNumber.touched">

</div>

<div class="form-row">

<div class="form-group col-md-6">

<label>SECURITY CODE</label>

<input type="password" class="form-control form-control-lg" placeholder="Security Code" name="securityCode"

#securityCode="ngModel" [(ngModel)]="service.formData.securityCode"

required maxlength="3" minlength="3" [class.invalid]="securityCode.invalid &#038;&#038; securityCode.touched">

</div>

<div class="form-group col-md-6">

<label>VALIDE THROUGH</label>

<input class="form-control form-control-lg" placeholder="MM/YY" name="expirationDate"

#expirationDate="ngModel" [(ngModel)]="service.formData.expirationDate"

required maxlength="5" minlength="5" [class.invalid]="expirationDate.invalid &#038;&#038; expirationDate.touched">

</div>

</div>

<div class="form-group">

<button class="btn btn-info btn-lg btn-block" type="submit" [disabled]="form.invalid">SUBMIT</button>

</div>

</form>

HTML

Copy

The above code snippet might be confusing for you because here we’ve put everything related to the form design and form validation. we have input fields for all model properties including paymentDetailId(hidden field). Each of the input field is bound to its respective property through 2 way data-binding.

Inside this form, all field has the required validation and number of characters is restricted to all field except CardOwnerName . While validating Angular form fields, we can use auto-generated classes/attributes for showing validation error indications. Auto-generated classes/attributes by Angular.

|  |  |  |
| --- | --- | --- |
| ng-invalid(class) or invalid(property) | X | ng-valid (class) or valid(property) |
| ng-untouched(class) or untouched(property) | X | ng-touched(class) or touched(property) |

To indicate validation error, we conditionally applied CSS class – invalid using the above properties invalid and touched. Finally, the submit button is conditionally disabled based on whether the form as a whole is valid or not.

Currently our Angular Form in payment-detail-form component looks like this.

## Insert a new Record

let’s implement submit event for the form.

<form ...

(submit)="onSubmit(form)">

...

</form>

HTML

Copy

Now define the function – onSubmit inside payment-detail-form.component.ts.

onSubmit(form: NgForm) {

this.insertRecord(form);

}

insertRecord(form: NgForm) {

this.service.postPaymentDetail().subscribe(

res => {

this.resetForm(form);

this.service.refreshList();

},

err => { console.log(err); }

)

}

TypeScript

Copy

A separate function insertRecord is defined to insert a new record into the SQL server table.

Before testing this operation, we have to **Enable CORS in Asp.Net Core API**, .Net Core Web API will block request from another application which is hosted in another domain or in another port number. by default, Angular is running at port number 4200 and Web API is hosted at a different port number. to make Http Request, we’ve to Enable-CORS (Cross Origin Resource Sharing) in Web API for the port 4200.

You just need to update Startup class like this.

public void ConfigureServices(IServiceCollection services)

{

...

//add cors package

services.AddCors();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

//configurations to cosume the Web API from port : 4200 (Angualr App)

app.UseCors(options =>

options.WithOrigins("http://localhost:4200")

.AllowAnyMethod()

.AllowAnyHeader());

...

}

C#

Copy

Inside the Configure function, it is better to keep the function call UseCors before any other lines. Now you can try the insert operation. for me, it is working fine. Comment if you face any problem.

## Retrieve and Display all Inserted Records

Inserted records can be retrieved and displayed in payment-details component, for that let’s update the component typescript file as follows.

import { PaymentDetailService } from './../../shared/payment-detail.service';

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

@Component({

selector: 'app-payment-details',

templateUrl: './payment-details.component.html',

styles: []

})

export class PaymentDetailsComponent implements OnInit {

constructor(public service: PaymentDetailService) { }

ngOnInit() {

this.service.refreshList();

}

}

TypeScript

Copy

Inside list component ngOnInit lifecycle hook, we’ve called refreshList function to populate list array in service class. using service list property, we can render all of the inserted records in payment-details component html. So update the table div as follow.

<table class="table">

<thead class="thead-light">

<tr>

<th>Card Owner</th>

<th>Card Number</th>

<th>Exp.</th>

<th></th>

</tr>

</thead>

<tr \*ngFor="let pd of service.list">

<td (click)="populateForm(pd)">{{pd.CardOwnerName}}</td>

<td (click)="populateForm(pd)">{{pd.CardNumber}}</td>

<td (click)="populateForm(pd)">{{pd.ExpirationDate}}</td>

<td>

<i class="far fa-trash-alt fa-lg text-danger" (click)="onDelete(pd.PMId)"></i>

</td>

</tr>

</table>

HTML

Copy

## Update and Delete Operation

To implement update operation, we’ve added click event for all td cells as shown below.

<td (click)="populateForm(pd)">...</td>

HTML

Copy

Inside the click event function, we have to populate the corresponding selected record inside the form. so add the following function to payment-details component.

populateForm(selectedRecord) {

this.service.formData = Object.assign({}, selectedRecord);

}

TypeScript

Inside the function, we just updated the selected record object into formData property in service class. since the form is bound to formData properties, the form field will get populated with correspondinAngular 11 CRUD with ASP.NET Core Web API

March 16, 2020 | ASP.NET Core Article | 55 Comments

Asp.Net Core Web API CRUD Operations with Angular 11

In this article, we’ll implement Asp.Net Core 5.0 Web API CRUD Operations with Angular 11. To demonstrate the topic, we’ll build a project from scratch with payment details like credit/ debit card.

GitHub repository: https://bit.ly/3qyiusO

Sub-topics discussed.

ASP.NET Core Web API

Create .NET Core Web API

Setup Database with EF Core

API Controller for CRUD Web Methods

Enable CORS for Angular App

Angular Client Side

Create Angular 11 Project

Consume ASP.NET Core API From Angular

Form Design and Validation

Insert/ Create Record by Form Submission

Retrieve and Display Inserted Records

Update and Delete Operation

Table of Contents

Create ASP.NET Core Web API

Setup Database

Create API Controller for CRUD Operations

Create Angular App

How to Consume .Net Core API from Angular

Form Design and Validation

Insert a new Record

Retrieve and Display all Inserted Records

Update and Delete Operation

Video Tutorial

Create ASP.NET Core Web API

In Visual Studio 2019, From the new project window, select Asp.Net Core Web Application.

Image showing how to create ASP.NET Core Web API project in Visual Studio.

Once you provide the project name and location. A new window will be opened as follows, Select API. The above steps will create a brand new ASP.NET Core Web API project.

Visual Studio Window showing creation of Asp.Net Core Web API Project.

Setup Database

Let’s create a Database for this project. Inside this project, we’ll be using Entity Framework Core to create and interact with the database. So first of all we’ve to install corresponding NuGet packages. Right-click on the project name from Solution Explorer, click on Manage NuGet Packages, from Browse Tab, install the following 3 packages with same version as that of Asp.Net Core.

Showing list of NuGet Packages for Entity Framework Core

Now, let’s define DB model class file –PaymentDetail.cs in a new folder Models.

public class PaymentDetail

{

[Key]

public int PaymentDetailId { get; set; }

[Required]

[Column(TypeName = "nvarchar(100)")]

public string CardOwnerName { get; set; }

[Required]

[Column(TypeName = "varchar(16)")]

public string CardNumber { get; set; }

[Required]

[Column(TypeName = "varchar(5)")]

public string ExpirationDate { get; set; }

[Required]

[Column(TypeName = "varchar(3)")]

public string SecurityCode { get; set; }

}

C#

Now let’s define DbContext class file- /Models/PaymentDetailContext.cs.

public class PaymentDetailContext : DbContext

{

public PaymentDetailContext(DbContextOptions<PaymentDetailContext> options):base(options)

{ }

public DbSet<PaymentDetail> PaymentDetails { get; set; }

}

C#

DbContext class- PaymentDetailContext decides what should be added to actual physical database during DB Migration. So we have added DbSet property for PaymentDetail Model class, after migration PaymentDetails table will be created in SQL Server Database.

Into this model class constructor parameter- options , we have to pass which DbProvider (SQL Server, MySQL, PostgreSQL, etc) to use and corresponding DB connection string also. For that, we’ll be using dependency injection in ASP.NET Core with Startup.cs file as follows.

public void ConfigureServices(IServiceCollection services)

{

...

services.AddDbContext<PaymentDetailContext>(optionns =>

optionns.UseSqlServer(Configuration.GetConnectionString("DevConnection")));

}

C#

Here we’ve used dependency injection for DbContext class, through which SQL Server is set as a DbProvider with a connection string, Now save the connection string in appsettings.json file using DevConnection key as follows.

{

....

"ConnectionStrings": {

"DevConnection": "Server=(local)\\sqlexpress;Database=PaymentDetailDB;Trusted\_Connection=True;MultipleActiveResultSets=True;"

}

}

JSON

Now let’s do the migration. Select project from solution explorer, then go to Tools > NuGet Package Manager > Package Manager Console. Then execute following commands one by one.

Add-Migration "InitialCreate"

Update-Database

none

After successful migration, as per the connection string, a new database – PaymentDetailDB will be created with PaymentDetails table. Also, there will be a new Migrations folder created with corresponding C# files.

Create API Controller for CRUD Operations

To create a new API controller, right-click on Controllers folder Add > Controller, Select API Controller with actions, using Entity Framework.

With the help of scaffolding mechanism, newly created PaymentDetailController will look like this.

[Route("api/[controller]")]

[ApiController]

public class PaymentDetailController : ControllerBase

{

private readonly PaymentDetailContext \_context;

public PaymentDetailController(PaymentDetailContext context)

{

\_context = context;

}

// GET: api/PaymentDetail

[HttpGet]

public async Task<ActionResult<IEnumerable<PaymentDetail>>> GetPaymentDetails()

{ ... }

// GET: api/PaymentDetail/5

[HttpGet("{id}")]

public async Task<ActionResult<PaymentDetail>> GetPaymentDetail(int id)

{ ... }

// PUT: api/PaymentDetail/5

[HttpPut("{id}")]

public async Task<IActionResult> PutPaymentDetail(int id, PaymentDetail paymentDetail)

{ ... }

// POST: api/PaymentDetail

[HttpPost]

public async Task<ActionResult<PaymentDetail>> PostPaymentDetail(PaymentDetail paymentDetail)

{ ... }

// DELETE: api/PaymentDetail/5

[HttpDelete("{id}")]

public async Task<ActionResult<PaymentDetail>> DeletePaymentDetail(int id)

{ ... }

private bool PaymentDetailExists(int id)

{ ... }

}

C#

It contains web methods POST, GET, PUT and DELETE for Create, Retrieve, Update and Delete operations respectively. As a constructor parameter we’ve context of the type PaymentDetailContext . the instance/value for this parameter will be passed from dependency injection from StartUp class.

For this project, we don’t have to change anything in web methods and you can test any of these CRUD operations using software like postman or you use open api support with the swagger interface.

Create Angular App

Now let’s create front-end client-side app in Angular 11. For that execute following Angular-CLI commands one by one.

ng new app\_name

# after project creation.

# navigate inside project folder

cd app\_name

# open in vs code

code .

# from vs code terminal

# command to open the app in default web browser

ng serve --o

none

Before moving forward, let’s look at the structure of the app that we want to build.

● src

+---● app

| +--● payment-details

| | |--payment-details.component.ts|.html

| | |

| | +--● payment-detail-form

| | |--payment-detail-form.component.ts|.html

| |

| +--● shared

| | |--payment-detail.service.ts

| | |--payment-detail.model.ts

| |

| |--app.module.ts

|

|--index.html (cdn path for bootstrap & fa icons)

None

We’ve two components, list of records will be shown in payment-details component, it has a child component payment-detail-form, inside that, a form for insert and update operation can be designed.

To create these 2 components, you can execute the following commands.

ng g c payment-details -s --skipTests

ng g c payment-details/payment-detail-form -s --skipTests

none

Options used

inlinestyle | -s : skip seperate component specific stylesheet

---skipTests : skip test files with extension .spec.ts

Now let’s replace the default component file- app.component.html as follows.

<div class="container">

<app-payment-details></app-payment-details>

</div>

HTML

To show list of records and it’s form side by side, update payment-details.component.html as bellow.

<div class="jumbotron py-3">

<h1 class="display-4 text-center">Payment Detail Register</h1>

</div>

<div class="row">

<div class="col-md-5">

<app-payment-detail-form></app-payment-detail-form>

</div>

<div class="col-md-7">

<div>table with list of records from the table</div>

</div>

</div>

HTML

For this app development, we’ll be using Bootstrap and Font Awesome Icons. so let’s add their stylesheet reference in index.html.

<!-- Add following stylesheets to <head> tag -->

<link rel="preload" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" data-rocket-async="style" as="style" onload="this.onload=null;this.rel='stylesheet'" integrity="sha384-JcKb8q3iqJ61gNV9KGb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN5t9UJ0Z" crossorigin="anonymous">

<link rel="preload" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.14.0/css/all.min.css" data-rocket-async="style" as="style" onload="this.onload=null;this.rel='stylesheet'" integrity="sha512-1PKOgIY59xJ8Co8+NE6FZ+LOAZKjy+KY8iq0G4B3CyeY6wYHN3yt9PW0XpSriVlkMXe40PTKnXrLnZ9+fkDaog==" crossorigin="anonymous" />

HTML

I’ve few custom CSS to add in global style sheet – styles.css.

.form-group label {

color: grey;

}

.form-group input {

font-weight: 500;

}

.form-group input::placeholder {

color: #c0bdbd;

font-weight: 300;

}

thead th{

font-weight: 400;

}

table tr:hover {

background-color: #fffbf2;

cursor: pointer;

}

/\*for invalid form controls\*/

input.invalid {

border-color: red;

}

CSS

How to Consume .Net Core API from Angular

First of all, let’s create a model class for payment details – shared/payment-detail.model.ts. You could manually create the file with the following CLI command.

ng g class shared/payment-detail --type=model --skipTests

# there is no seperat command for model

# hence we use class creation command with type option

# type is used to name the file like '.model.ts'

none

Update the model class with corresponding properties similar to .Net Core API model properties.

export class PaymentDetail {

paymentDetailId: number=0;

cardOwnerName: string='';

cardNumber: string='';

expirationDate: string='';

securityCode: string='';

}

C#

Now let’s create a service class to interact with ASP.NET Core Web API- shared/payment-detail.service.ts. Here is the CLI command to create the service class.

ng g s shared/payment-detail --skipTests

none

Update the service class as below.

import { PaymentDetail } from './payment-detail.model';

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

import { HttpClient } from "@angular/common/http";

@Injectable({

providedIn: 'root'

})

export class PaymentDetailService {

formData: PaymentDetail= new PaymentDetail();

readonly baseURL = 'http://localhost:61236/api/PaymentDetail';

list : PaymentDetail[];

constructor(private http: HttpClient) { }

postPaymentDetail() {

return this.http.post(this.baseURL, this.formData);

}

putPaymentDetail() {

return this.http.put(`${this.baseURL}/${this.formData.paymentDetailId}`, this.formData);

}

deletePaymentDetail(id: number) {

return this.http.delete(`${this.baseURL}/${id}`);

}

refreshList() {

this.http.get(this.baseURL)

.toPromise()

.then(res =>this.list = res as PaymentDetail[]);

}

}

JavaScript

formData property can be used for designing the form for CRUD Operations, list array is used to store all of the records from the API. baseURL contains the base URL for the Web API controller. Now let’s run the API from Visual Studio – Debug > Start Debugging(F5). HttpClient is used to make HTTP request to the server. Along with methods for CRUD operations, we’ve refreshList function to populate existing records into list property.

To use HttpClient , we also need to import HttpClientModule . So update app/app.module.ts as follows.

...

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

@NgModule({

...

imports: [HttpClientModule, ...],

...

})

TypeScript

Form Design and Validation

Now let’s design the form in payment-detail-form component.

import { PaymentDetailService } from './../../shared/payment-detail.service';

import { PaymentDetail } from 'src/app/shared/payment-detail.model';

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

import { NgForm } from '@angular/forms';

@Component({

selector: 'app-payment-detail-form',

templateUrl: './payment-detail-form.component.html',

styles: []

})

export class PaymentDetailFormComponent implements OnInit {

constructor(public service: PaymentDetailService) { }

ngOnInit():void {

}

resetForm(form: NgForm) {

form.form.reset();

this.service.formData = new PaymentDetail();

}

}

TypeScript

Since we’ve the service injected here. we can access it’s property formData in component HTML file for designing the form. resetForm function will reset the form to its initiale state.

As a first step towards designing the form, we’ve to import FormsModule in app/app.module.ts.

...

import { FormsModule } from '@angular/forms';

@NgModule({

...

imports: [FormsModule, ...],

...

})

TypeScript

Now let’s design the form. so update payment-detail-form.component.html as shown below.

<form novalidate autocomplete="off" #form="ngForm" (submit)="onSubmit(form)">

<input type="hidden" name="paymentDetailId" [value]="service.formData.paymentDetailId" />

<div class="form-group">

<label>CARD OWNER NAME</label>

<input class="form-control form-control-lg" placeholder="Full Name" name="cardOwnerName"

#cardOwnerName="ngModel" [(ngModel)]="service.formData.cardOwnerName"

required [class.invalid]="cardOwnerName.invalid &#038;&#038; cardOwnerName.touched">

</div>

<div class="form-group">

<label>CARD NUMBER</label>

<input class="form-control form-control-lg" placeholder="16 Digit Card Number" name="cardNumber"

#cardNumber="ngModel" [(ngModel)]="service.formData.cardNumber"

required maxlength="16" minlength="16" [class.invalid]="cardNumber.invalid &#038;&#038; cardNumber.touched">

</div>

<div class="form-row">

<div class="form-group col-md-6">

<label>SECURITY CODE</label>

<input type="password" class="form-control form-control-lg" placeholder="Security Code" name="securityCode"

#securityCode="ngModel" [(ngModel)]="service.formData.securityCode"

required maxlength="3" minlength="3" [class.invalid]="securityCode.invalid &#038;&#038; securityCode.touched">

</div>

<div class="form-group col-md-6">

<label>VALIDE THROUGH</label>

<input class="form-control form-control-lg" placeholder="MM/YY" name="expirationDate"

#expirationDate="ngModel" [(ngModel)]="service.formData.expirationDate"

required maxlength="5" minlength="5" [class.invalid]="expirationDate.invalid &#038;&#038; expirationDate.touched">

</div>

</div>

<div class="form-group">

<button class="btn btn-info btn-lg btn-block" type="submit" [disabled]="form.invalid">SUBMIT</button>

</div>

</form>

HTML

The above code snippet might be confusing for you because here we’ve put everything related to the form design and form validation. we have input fields for all model properties including paymentDetailId(hidden field). Each of the input field is bound to its respective property through 2 way data-binding.

Inside this form, all field has the required validation and number of characters is restricted to all field except CardOwnerName . While validating Angular form fields, we can use auto-generated classes/attributes for showing validation error indications. Auto-generated classes/attributes by Angular.

ng-invalid(class) or

invalid(property) X ng-valid (class) or

valid(property)

ng-untouched(class) or

untouched(property) X ng-touched(class) or

touched(property)

To indicate validation error, we conditionally applied CSS class – invalid using the above properties invalid and touched. Finally, the submit button is conditionally disabled based on whether the form as a whole is valid or not.

Currently our Angular Form in payment-detail-form component looks like this.

Image showing Angular Form design

Insert a new Record

let’s implement submit event for the form.

<form ...

(submit)="onSubmit(form)">

...

</form>

HTML

Now define the function – onSubmit inside payment-detail-form.component.ts.

onSubmit(form: NgForm) {

this.insertRecord(form);

}

insertRecord(form: NgForm) {

this.service.postPaymentDetail().subscribe(

res => {

this.resetForm(form);

this.service.refreshList();

},

err => { console.log(err); }

)

}

TypeScript

A separate function insertRecord is defined to insert a new record into the SQL server table.

Before testing this operation, we have to Enable CORS in Asp.Net Core API, .Net Core Web API will block request from another application which is hosted in another domain or in another port number. by default, Angular is running at port number 4200 and Web API is hosted at a different port number. to make Http Request, we’ve to Enable-CORS (Cross Origin Resource Sharing) in Web API for the port 4200.

You just need to update Startup class like this.

public void ConfigureServices(IServiceCollection services)

{

...

//add cors package

services.AddCors();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

//configurations to cosume the Web API from port : 4200 (Angualr App)

app.UseCors(options =>

options.WithOrigins("http://localhost:4200")

.AllowAnyMethod()

.AllowAnyHeader());

...

}

C#

Inside the Configure function, it is better to keep the function call UseCors before any other lines. Now you can try the insert operation. for me, it is working fine. Comment if you face any problem.

Retrieve and Display all Inserted Records

Inserted records can be retrieved and displayed in payment-details component, for that let’s update the component typescript file as follows.

import { PaymentDetailService } from './../../shared/payment-detail.service';

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

@Component({

selector: 'app-payment-details',

templateUrl: './payment-details.component.html',

styles: []

})

export class PaymentDetailsComponent implements OnInit {

constructor(public service: PaymentDetailService) { }

ngOnInit() {

this.service.refreshList();

}

}

TypeScript

Inside list component ngOnInit lifecycle hook, we’ve called refreshList function to populate list array in service class. using service list property, we can render all of the inserted records in payment-details component html. So update the table div as follow.

<table class="table">

<thead class="thead-light">

<tr>

<th>Card Owner</th>

<th>Card Number</th>

<th>Exp.</th>

<th></th>

</tr>

</thead>

<tr \*ngFor="let pd of service.list">

<td (click)="populateForm(pd)">{{pd.CardOwnerName}}</td>

<td (click)="populateForm(pd)">{{pd.CardNumber}}</td>

<td (click)="populateForm(pd)">{{pd.ExpirationDate}}</td>

<td>

<i class="far fa-trash-alt fa-lg text-danger" (click)="onDelete(pd.PMId)"></i>

</td>

</tr>

</table>

HTML

Update and Delete Operation

To implement update operation, we’ve added click event for all td cells as shown below.

<td (click)="populateForm(pd)">...</td>

HTML

Inside the click event function, we have to populate the corresponding selected record inside the form. so add the following function to payment-details component.

populateForm(selectedRecord) {

this.service.formData = Object.assign({}, selectedRecord);

}

TypeScript

Inside the function, we just updated the selected record object into formData property in service class. since the form is bound to formData properties, the form field will get populated with corresponding details.

After making required changes in these populated value fields, user can submit the form for the update operation. so we have to handle both insert and update operation inside the same form submit event in payment-detail-form component. hence update the payment-detail-form.component.ts.

onSubmit(form: NgForm) {

if (this.service.formData.PMId == 0)

this.insertRecord(form);

else

this.updateRecord(form);

}

updateRecord(form: NgForm) {

this.service.putPaymentDetail().subscribe(

res => {

this.resetForm(form);

this.service.refreshList();

},

err => {

console.log(err);

}

)

}

TypeScript

Inside the form, we have a hidden field for paymentDetailId. Based on its value in submit function, we can decide whether we’ve got an insert/ update operation. insertRecord is already defined. with updateRecord function, we will update the corresponding payment-detail record.

Now let’s implement Delete Operation in parent list component. For that add delete button inside that table rows.

update payment-details.component.html as shown below.

...

<td>

<i class="far fa-trash-alt fa-lg text-danger" (click)="onDelete(pd.paymentDetailId)"></i>

</td>

</tr>

</table>

HTML

Now lets define onDelete function in payment-details.component.ts.

onDelete(PMId) {

if (confirm('Are you sure to delete this record ?')) {

this.service.deletePaymentDetail(id: number)

.subscribe(res => {

this.service.refreshList();

},

err => { console.log(err); })

}

}

TypeScript

So that’s all about Angular CRUD operations with ASP.NET Core Web API.g details.

After making required changes in these populated value fields, user can submit the form for the update operation. so we have to handle both insert and update operation inside the same form submit event in payment-detail-form component. hence update the payment-detail-form.component.ts.

onSubmit(form: NgForm) {

if (this.service.formData.PMId == 0)

this.insertRecord(form);

else

this.updateRecord(form);

}

updateRecord(form: NgForm) {

this.service.putPaymentDetail().subscribe(

res => {

this.resetForm(form);

this.service.refreshList();

},

err => {

console.log(err);

}

)

}

TypeScript

Copy

Inside the form, we have a hidden field for paymentDetailId. Based on its value in submit function, we can decide whether we’ve got an insert/ update operation. insertRecord is already defined. with updateRecord function, we will update the corresponding payment-detail record.

Now let’s implement Delete Operation in parent list component. For that add delete button inside that table rows.

update payment-details.component.html as shown below.

...

<td>

<i class="far fa-trash-alt fa-lg text-danger" (click)="onDelete(pd.paymentDetailId)"></i>

</td>

</tr>

</table>

HTML

Copy

Now lets define onDelete function in payment-details.component.ts.

onDelete(PMId) {

if (confirm('Are you sure to delete this record ?')) {

this.service.deletePaymentDetail(id: number)

.subscribe(res => {

this.service.refreshList();

},

err => { console.log(err); })

}

}

So that’s all about Angular CRUD operations with ASP.NET Core Web API.