Angular 9 CRUD with ASP.NET Core Web API

33 SHARES Share Tweet Linkedin Subscribe Whatsapp +

In this article, we'll implement **Asp.Net Core 3.0 Web API CRUD Operations with Angular 9**. To demonstrate the topic, we'll build a project from scratch with payment details like Credit/ Debit Card.

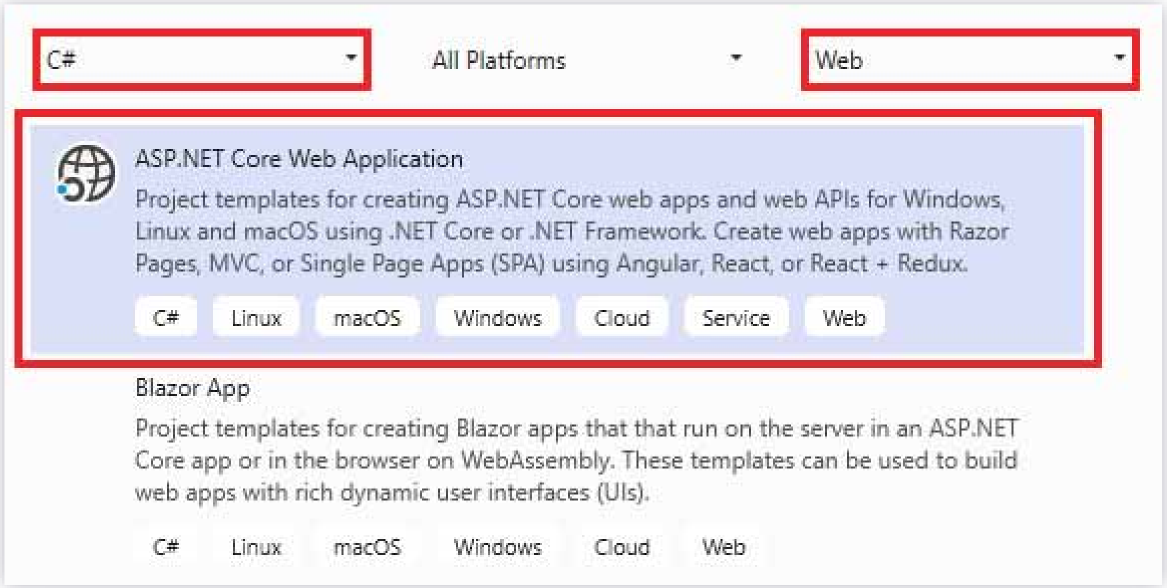
Sub-topics discussed.

- ASP.NET Core Web API
  - Create .Net Core Web API
  - Setup Database with EF Core
  - API Controller for CRUD Web Methods
- Angular Client Side
  - Create Angular 9 Project
  - Consume ASP.NET Core API From Anguar
  - Form Design and Validation
  - Insert/ Create Record by Form Submission

- Retrive and Display Inserted Records
- Update and Delete Operation

Create ASP.NET Core Web API

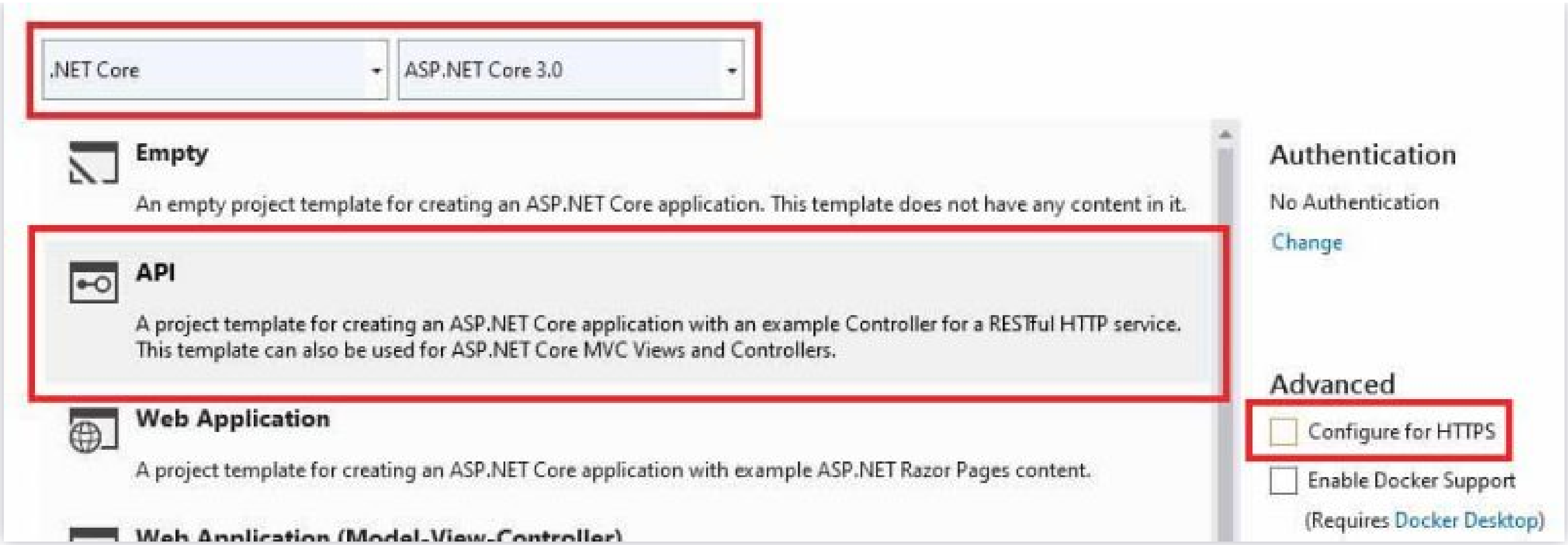
In Visual Studio 2019, Go to File > New > Project (Ctrl + Shift + N).
From *New Project* window, Select *Asp.Net Core Web Application.*


C#
All Platforms
Web

ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.
C# Linux macOS Windows Cloud Service Web

Blazor App
Project templates for creating Blazor apps that that run on the server in an ASP.NET Core app or in the browser on WebAssembly. These templates can be used to build web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

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


.NET Core
ASP.NET Core 3.0

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Authentication
No Authentication
Change

Advanced
☐ Configure for HTTPS
☐ Enable Docker Support
(Requires Docker Desktop)

Setup Database

Let's create a Database for this project. Inside this project, we'll be using Entity Framework Core to do DB Operations. So first of all we've to install corresponding NuGet packages. Right click on Project Name from *Solution Explorer*, Click on *Manage NuGet Packages,* In *Browse Tab*– install following 3 packages.

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

```
public class PaymentDetail
{
    [Key]
    public int PMId { 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 CVV { get; set; }
}
```

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; }
}
```

DbContext Class- *PaymentDetailContext* decides what should be added to actual physical DB 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 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")));
}
```

Here we've used Dependency Injection for DbContext class, by passing SQL Server as a DbProvider with Connection String, Now let's save connections in *appsettings.json* file using *DevConnection* key as follows.

```
{

  ....

  "ConnectionStrings": {
    "DevConnection": "Server=(local)\\sqlexpress;Database=PaymentDetailDB;Trusted_Connection=True;MultipleActiveRe
    }
  }
```

We’ve done everything for Database, Now let’s do the migration. For that 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
```

After successful migration, As per the connection string, new Database – PaymentDetailDB will be created with *PaymentDetails* table. Also there will be new *Migrations* folder in Solution Explorer 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.* then we can create *PaymentDetailController* for CRUD operations.

With the help of Scaffolding Mechanism, new controller will be created.

```
[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)
        { ... }
    }
```

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 in *StartUp* class.

For this project, we don't have to change anything in web methods. You can test any of the CRUD operations using softwares like postman. Web methods with corresponding URL is given below.

GET	/api/PaymentDetail/	Retrieve all records
GET	/api/PaymentDetail/id	Retrieve a record with given id
POST	/api/PaymentDetail/	Insert/ Create a new record
PUT	/api/PaymentDetail/id	Update a record with given id
DELETE	/api/PaymentDetail/id	Delete a record with given id

Create Angular App

Now let's create front-end client-side app in Angular 9. For that execute following Angular-CLI command.

```
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
```

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|.css
|   |  +--● payment-detail
|   |  |  |--payment-detail.component.ts|.html|.css
|   |  |
|   |  +--● payment-detail-list
|   |  |  |--payment-detail-list.component.ts|.html|.css
|   |  |
|   |  +--● shared
|   |     |--payment-detail.service.ts
|   |     |--payment-detail.model.ts
|   |
|   |--app.module.ts
|
|--index.html (cdn path for bootstrap & fa icons)
```

As parent component, we’ve *payment-details.* it has two child components

1. payment-detail (Form design and related operations – insert and update)
2. payment-detail-list (Retrieve and display existing records)

to create these 3 component, you can execute following commands.

```
//parent component
ng g c payment-details -s --skipTests
//child components
ng g c payment-details/payment-detail -s --skipTests
ng g c payment-details/payment-detail-list -s --skipTests
```

> *options –inlineStyle(Aliase : -s) and –skipTests are used to skip seperate style sheet and test(extension – .spec.ts) files respectively.*

now let’s replace the default component html(app.component.html) as follows.

```
<div class="container">
  <app-payment-details></app-payment-details>
</div>
```

to show child components side by side update parent component html as below. -payment-details.component.html

```
<div class="jumbtron">
  <h1 class="display-4 text-center">Payment Detail Register</h1>
  <hr>
  <div class="row">
    <div class="col-md-5">
```

```
        <app-payment-detail></app-payment-detail>
      </div>
      <div class="col-md-7">
        <app-payment-detail-list></app-payment-detail-list>
      </div>
    </div>
  </div>
```

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

```
<head>
    ...
    <link rel="preload" href="https://stackpath.bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.css" as="style"
    <link rel="preload" href="https://use.fontawesome.com/releases/v5.6.3/css/all.css" as="style" onload="this.onl
</head>
...
```

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

```
input.ng-touched.ng-invalid{
    border-color: #dc3545;
}

input.ng-valid{
    border-color: #28a745;
}
```

```
.green-icon{
    color: #28a745;
}
.red-icon{
    color: #dc3545;
}

#toast-container > div {
    opacity:1;
}

table tr:hover{
    cursor: pointer;
}
```

How to Consume .Net Core API from Angular

first of let’s create model class for PaymentDetails – shared/payment-detail.model.ts. You could manually create the file or execute 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'
```

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

```
export class PaymentDetail {
    PMId :number;
    CardOwnerName: string;
    CardNumber: string;
    ExpirationDate: string;
    CVV: string;
}
```

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
```

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;
  readonly rootURL = 'http://localhost:65067/api';
  list : PaymentDetail[];

  constructor(private http: HttpClient) { }

  postPaymentDetail() {
    return this.http.post(this.rootURL + '/PaymentDetail', this.formData);
  }
  putPaymentDetail() {
    return this.http.put(this.rootURL + '/PaymentDetail/'+ this.formData.PMId, this.formData);
  }
  deletePaymentDetail(id) {
    return this.http.delete(this.rootURL + '/PaymentDetail/'+ id);
  }

  refreshList(){
    this.http.get(this.rootURL + '/PaymentDetail')
    .toPromise()
    .then(res => this.list = res as PaymentDetail[]);
  }
}
```

formData property can be used to design the form for CRUD Operations, *list* array is used to store all of the retrieved records from the API. *rootUrl* contains the base URL of the Web API. You've to 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.*

This service class must be injected at root level, that's why it has *Injectable* decorator with *providedIn* as *root.* to inject this class at root level and to use *HttpClient,* we also need to import *HttpClientModule.* So update app/app.module.ts as follows.

```
...
import { HttpClientModule } from '@angular/common/http';
import { PaymentDetailService } from './shared/payment-detail.service';

@NgModule({
  ...
  imports: [HttpClientModule, ...],
  providers: [PaymentDetailService],
  ...
})
```

Form Design and Validation

Now let's design the form in *payment-detail* component. first of all we've to inject the service class.

```
import { PaymentDetailService } from './../../shared/payment-detail.service';
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';

```

```
@Component({
  selector: 'app-payment-detail',
  templateUrl: './payment-detail.component.html',
  styles: []
})
export class PaymentDetailComponent implements OnInit {

  constructor(private service: PaymentDetailService) { }

  ngOnInit() {
    this.resetForm();
  }

  resetForm(form?: NgForm) {
    if (form != null)
      form.form.reset();
    this.service.formData = {
      PMId: 0,
      CardOwnerName: '',
      CardNumber: '',
      ExpirationDate: '',
      CVV: ''
    }
  }
}
```

Since we've the service injected here. we can access the service property *formData* to design the form. function *resetForm* can be used to initialize model property or reset form based on parameter *form.* Inside *ngOnInit* life-cycle event, we've called the function to initialize the model property.

first of all, we’ve to import *FormsModule* in app/*app.module.ts.*

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

@NgModule({
  ...
  imports: [FormsModule, ...],
  ...
})
```

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

```
<form #form="ngForm" autocomplete="off">
  <input type="hidden" name="PMId" [value]="service.formData.PMId">
  <div class="form-group">
    <div class="input-group">
      <div class="input-group-prepend">
        <div class="input-group-text bg-white">
          <i class="fas fa-user-circle" [class.green-icon]="CardOwnerName.valid" [class.red-icon]="CardOwnerName.i
        </div>
      </div>
      <input name="CardOwnerName" #CardOwnerName="ngModel" [(ngModel)]="service.formData.CardOwnerName" class="for
        placeholder="Card Owner Name" required>
    </div>
  </div>
  <div class="form-group">
    <div class="input-group">
```

```
        <div class="input-group-prepend">
          <div class="input-group-text bg-white">
            <i class="far fa-credit-card"  [class.green-icon]="CardNumber.valid" [class.red-icon]="CardNumber.invali
          </div>
        </div>
        <input name="CardNumber" #CardNumber="ngModel" [(ngModel)]="service.formData.CardNumber"
        class="form-control" placeholder="16 Digit Card Number" required maxlength="16" minlength="16">
      </div>
    </div>
    <div class="form-row">
      <div class="form-group col-md-7">
        <div class="input-group">
          <div class="input-group-prepend">
            <div class="input-group-text bg-white">
              <i class="fas fa-calendar-alt"  [class.green-icon]="ExpirationDate.valid" [class.red-icon]="Expiration
            </div>
          </div>
          <input name="ExpirationDate" #ExpirationDate="ngModel" [(ngModel)]="service.formData.ExpirationDate" class
            placeholder="MM/YY" required maxlength="5" minlength="5">
        </div>
      </div>
      <div class="form-group col-md-5">
        <div class="input-group">
          <div class="input-group-prepend">
            <div class="input-group-text bg-white">
              <i class="fas fa-key"  [class.green-icon]="CVV.valid" [class.red-icon]="CVV.invalid && CVV.touched"></
            </div>
          </div>
          <input type="password" name="CVV" #CVV="ngModel" [(ngModel)]="service.formData.CVV" class="form-control" p
          required  maxlength="3" minlength="3">
        </div>
      </div>
    </div>
    <div class="form-group">
      <button class="btn btn-success btn-lg btn-block" type="submit" [disabled]="form.invalid"><i class="fas fa-datal
    </div>
  </form>
```

it might be confusing for you, because here we've put everything related to the form – design and form validation. we have input field for all model properties including *PMId* in a hidden field. Each 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 *Card Owner Name.* For Angular field validation, we can use angular classes/attributes for showing validation error indications. auto-generated classes/ attribute by Angular

- ng-invalid (class)/ invalid (property) X ng-valid (class)/ valid (property)
- ng-untouched (class)/ untouched (property) X ng-touched (class)/ touched (property)

for each input field we've a Font Awesome Icon. To indicate validation error, we conditionally change font color of these icons using css class *green-icon* and *red-icon.* 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* component looks like this.

Insert a new Record

let’s wire up *submit* event to the form.

```
<form ...
    (submit)="onSubmit(form)">
...
</form>
```

Now define the function – onSubmit inside *payment-detail.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); }
  )
}
```
a separate function *insertRecord* is defined to insert a new record into the SQL server table.

Before testing this operation, we have to do a few more things in ASP.NET Core Web API.

1. .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.
2. By default, ASP.Net Core API use camel casing for response object. (eg: **C**ardNumber to **c**ardNumber). so we've to avoid this default json formatting.

Now let's do required steps to solve these two issues. First of all install latest NuGet Package – *Microsoft.AspNetCore.Cors.* then you can update startup class like this.

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    //remove default json formatting
    services.AddControllers().AddJsonOptions(options =>
     {
         options.JsonSerializerOptions.PropertyNamingPolicy = null;
         options.JsonSerializerOptions.DictionaryKeyPolicy = null;
     });

    //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());

    ...
}
```

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-detail-list* component. for that let's update the component *ts* file as follows.

```
import { PaymentDetailService } from './../../shared/payment-detail.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-payment-detail-list',
  templateUrl: './payment-detail-list.component.html',
  styles: []
})
export class PaymentDetailListComponent implements OnInit {

  constructor(pubic service: PaymentDetailService) { }

  ngOnInit() {
    this.service.refreshList();
  }
}
```

Inside list component- *ngOnInit* life-cycle 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 list component html.

```
<table class="table table-hover">
    <tr *ngFor="let pd of service.list">
        <td>{{pd.CardOwnerName}}</td>
        <td>{{pd.CardNumber}}</td>
        <td>{{pd.ExpirationDate}}</td>
    </tr>
</table>
```

Update and Delete Operation

Now let's implement Update Operation, for that we can add click event for all *td* cells as shown below.

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

inside the click event funtion, we have to populate corresponding selected record inside the form. so add following function to *payment-detail-list* component.

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

inside the function, we just set the selected record object to *formData* property in service class. since the form is bound to *formData* properties, the form field will get populated corresponding details.

After making required changes in these populated value fields, user can submit the form for the update operation. so we have handle both insert and update operation inside the form submit event in *payment-detail* component. hence update the *payment-detail.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.toastr.info('Submitted successfully', 'Payment Detail Register');
      this.service.refreshList();
    },
    err => {
      console.log(err);
    }
  )
}
```

inside the form, we have a hidden field for *PMId* 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 list component. for that add a button to delete a record inside that table.

file : payment-detail-list.component.html

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

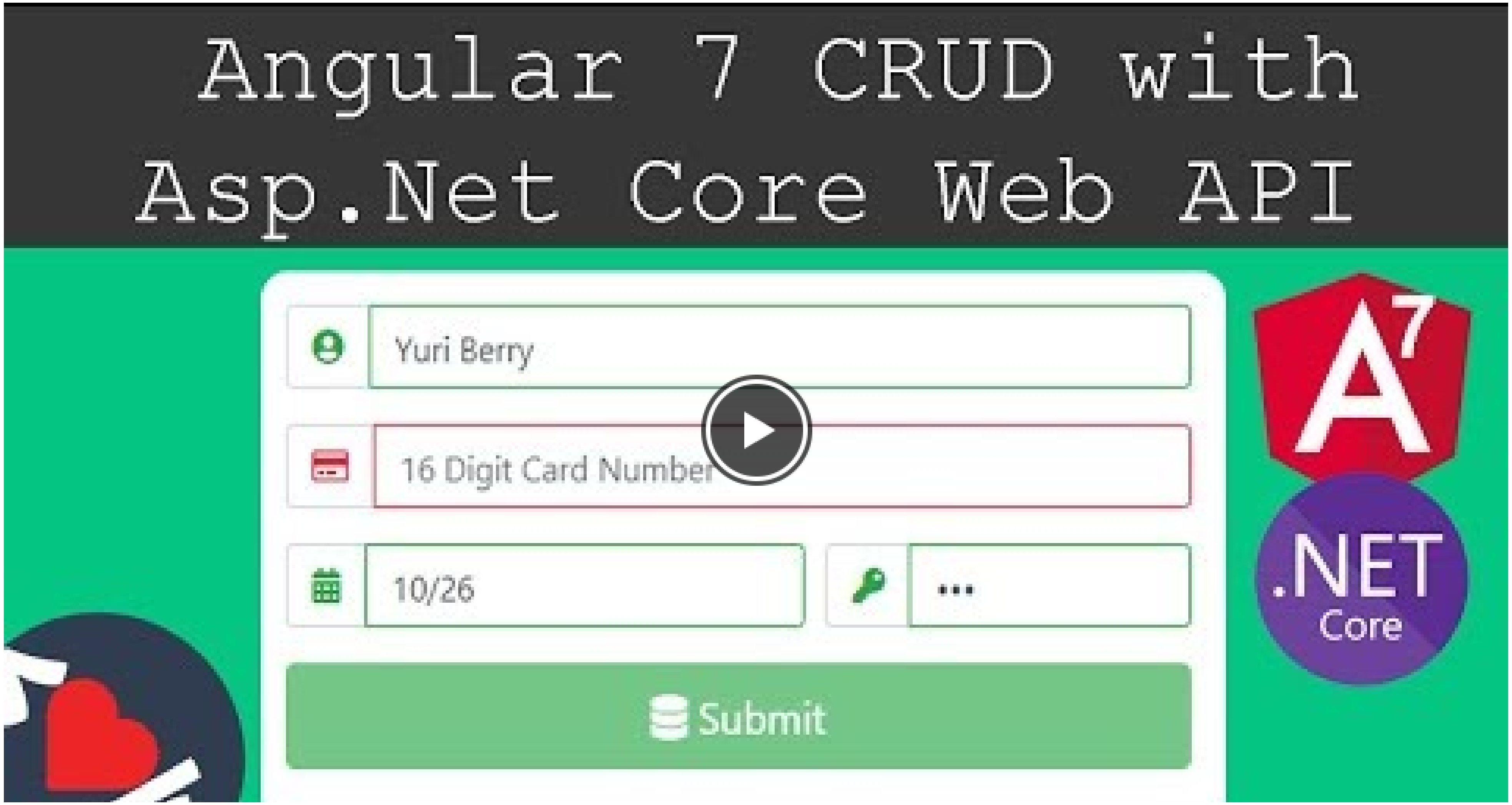
now let define *onDelete* function in *payment-detail-list.component.ts* as below.

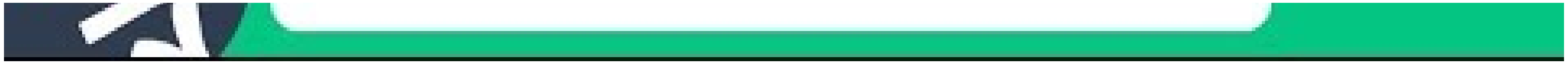
```
onDelete(PMId) {
  if (confirm('Are you sure to delete this record ?')) {
    this.service.deletePaymentDetail(PMId)
      .subscribe(res => {
        this.service.refreshList();
      },
      err => { console.log(err); })
  }
}
```

So in this article, we’ve completed Angular CRUD Operations With ASP.NET Core Web API.

Step By Step Video Tutorial

In our YouTube channel, we did these CRUD Operations with Asp.Net Core Web API 2.2 and Angular 7.

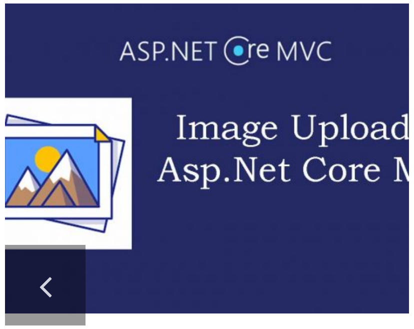




33 SHARES | Share | Tweet | Linkedin | Subscribe | Whatsapp | +

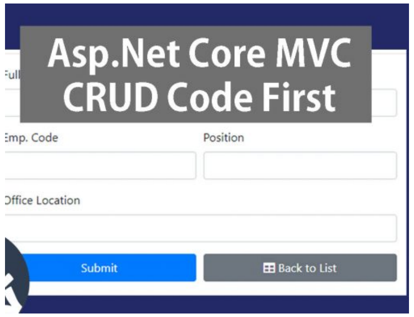
March 16th, 2020 | Angular, Angular Article, ASP.NET Core, ASP.NET Core Article | 3 Comments

Related Posts



ASP.NET Core MVC Image Upload and Retrieve

March 23rd, 2020 | 0 Comments



Asp.Net Core MVC CRUD with EF Core

March 14th, 2019 | 7 Comments

Angular 7 CRUD with Firestore

November 12th, 2018 | 20 Comments

3 Comments

James March 16, 2020 at 11:30 pm - Reply

Good article. Might I suggest introducing string interpolation though as it seems to be the status quo. For instance:
putPaymentDetail() {
return this.http.put(this.rootURL + '/PaymentDetail/'+ this.formData.PMId, this.formData);
}

Could be
putPaymentDetail() {
return this.http.put(`this.rootURL/${PaymentDetail}/${this.formData.PMId}`, this.formData);
}

Shamseer March 17, 2020 at 6:08 am - Reply

thanks for the info.

Mahender Reddy March 18, 2020 at 9:51 pm - Reply

It is very nice content , which is also more useful.
i am requesting to perform one more crud operations on Ag-Grid in angular -9 , as there is no such content in web.
Also puling users from active directory , if gives us with good examples it will be very much useful , as there are no such examples on web.

Thanks

Leave A Comment

Comment...

Name (required)

Email (required)

Website

Post Comment

Stay Connected

2,481 Fans

71,022 Subscribers

134 Followers

PayPal Donation

Popular | Recent

User Registration Form in MEAN Stack Using Angular 6 – Front End

July 23rd, 2018

How to Create Asp.net Mvc Application

May 14th, 2017

Joins in Sql Server

May 1st, 2017

Routing in Asp.Net MVC

June 1st, 2017

JWT Authentication in Node JS With MEAN Stack Application

August 7th, 2018

Subcribe to Our Blog via Email

Join 2,324 Subscribers

Email Address

Subscribe

Search...

Most Shared Posts

Asp.Net Core MVC CRUD with EF Core...

84 Shares

Angular 5 with Web API – CRUD Operations...

57 Shares

Angular 5 Login and Logout with Web API Using Toke...

46 Shares

MEAN Stack Login and Logout with Angular 6...

35 Shares

Angular 9 CRUD with ASP.NET Core Web API...

33 Shares

Angular 5 User Registration with Web API...

21 Shares