***Create a table named Customer\_review in mssql server and perform crud operation through apis with entityframework, microsoft design and tools***

Set Up Entity Framework in Your Project

dotnet add package Microsoft.EntityFrameworkCore dotnet add package Microsoft.EntityFrameworkCore.SqlServer dotnet add package Microsoft.EntityFrameworkCore.Tools

5. Configure the PortfolioContext in Startup.cs or Program.cs

In Startup.cs or Program.cs, register the database context:

csharp

Copy code

services.AddDbContext<PortfolioContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("PortfolioDb")));

6. Create and Apply Migrations

Run the following commands to create and apply the initial migration:

bash

Copy code

dotnet ef migrations add InitialCreate

dotnet ef database update

7. Create CRUD Endpoints in a Controller

Create a new controller, CustomerReviewController, and add CRUD actions:

csharp

Copy code

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

[ApiController]

public class CustomerReviewController : ControllerBase

{

private readonly PortfolioContext \_context;

public CustomerReviewController(PortfolioContext context)

{

\_context = context;

}

[HttpGet]

public async Task<ActionResult<IEnumerable<CustomerReview>>> GetReviews()

{

return await \_context.CustomerReviews.ToListAsync();

}

[HttpGet("{id}")]

public async Task<ActionResult<CustomerReview>> GetReview(int id)

{

var review = await \_context.CustomerReviews.FindAsync(id);

if (review == null) return NotFound();

return review;

}

[HttpPost]

public async Task<ActionResult<CustomerReview>> CreateReview(CustomerReview review)

{

\_context.CustomerReviews.Add(review);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetReview), new { id = review.Id }, review);

}

[HttpPut("{id}")]

public async Task<IActionResult> UpdateReview(int id, CustomerReview review)

{

if (id != review.Id) return BadRequest();

\_context.Entry(review).State = EntityState.Modified;

await \_context.SaveChangesAsync();

return NoContent();

}

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteReview(int id)

{

var review = await \_context.CustomerReviews.FindAsync(id);

if (review == null) return NotFound();

\_context.CustomerReviews.Remove(review);

await \_context.SaveChangesAsync();

return NoContent();

}

}

8. Test Your API Endpoints

Use tools like Postman or Swagger to test the CRUD operations.

**Improvement**

**Program.cs**

**Strengths:**

* Properly registers services for PortfolioContext and repositories.
* CORS policy is configured to allow requests from http://localhost:4200.
* Middleware is set up correctly (Swagger, UseCors, UseStaticFiles, UseHttpsRedirection, etc.).

**Suggestions:**

1. **Consider using dependency injection for the connection string**: Instead of directly fetching the connection string from builder.Configuration, you can make it more flexible:

csharp

Copy code

builder.Services.AddDbContext<PortfolioContext>(options =>

options.UseSqlServer(builder.Configuration.GetSection("ConnectionStrings:PortfolioDb").Value));

1. **Add Exception Handling Middleware**: Add middleware to handle unhandled exceptions globally:

csharp

Copy code

if (app.Environment.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

}

1. **Ensure Logging is Configured**: Make sure logging is configured in appsettings.json for better debugging and monitoring.

**CustomerReviewController**

**Strengths:**

* Constructor injection is properly implemented for the ICustomerReviewService.

**Suggestions:**

1. **Use Fluent API Routing**: Instead of [Route("api/[controller]")], you can explicitly name the route for better readability and to avoid potential issues with pluralization:

csharp

Copy code

[Route("api/customer-reviews")]

1. **Add Documentation Comments**: Provide XML documentation for each action method to help developers understand their purpose.
2. **Error Handling**: Add try-catch blocks in the controller for API-specific error responses (e.g., BadRequest, NotFound):

csharp

Copy code

[HttpGet]

public async Task<IActionResult> GetAllReviews()

{

try

{

var reviews = await \_customerReviewService.GetAllReviewsAsync();

return Ok(reviews);

}

catch (Exception ex)

{

return StatusCode(500, "An error occurred while retrieving reviews.");

}

}

**CustomerReviewService**

**Strengths:**

* Dependencies are well-separated using interfaces.
* Logger is properly injected for tracking operations.

**Suggestions:**

1. **Consider Using Unit of Work**: If \_unitOfWork was meant to coordinate repository operations, reintroduce it to ensure consistency across multiple database operations. For example:

csharp

Copy code

public class CustomerReviewService : ICustomerReviewService

{

private readonly IUnitOfWork \_unitOfWork;

private readonly ILogger<CustomerReview> \_logger;

public CustomerReviewService(IUnitOfWork unitOfWork, ILogger<CustomerReview> logger)

{

\_unitOfWork = unitOfWork;

\_logger = logger;

}

}

1. **Validation**: Add validation logic to ensure that incoming data (e.g., a new review) meets required conditions before persisting it to the database.

**CustomerReviewRepository**

**Strengths:**

* Makes use of PortfolioContext for database access.
* Includes asynchronous methods (async/await), which is a best practice.

**Suggestions:**

1. **Add Logging**: Log operations like data retrieval, addition, or updates for traceability:

csharp

Copy code

public async Task<List<CustomerReview>> GetAllReviewsAsync()

{

\_logger.LogInformation("Fetching all customer reviews.");

return await \_context.CustomerReviews.ToListAsync();

}

1. **Consider a Generic Repository Pattern**: If you have multiple repositories, a generic repository can reduce redundancy:

csharp

Copy code

public interface IRepository<T> where T : class

{

Task<List<T>> GetAllAsync();

Task<T> GetByIdAsync(int id);

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(int id);

}

**FileService and FileRepository**

**Strengths:**

* Separation of concerns is properly maintained.
* File repository uses a specific folder (wwwroot/assets/images) for storing files.

**Suggestions:**

1. **File Path Validation**: Ensure the file path is validated to prevent directory traversal attacks:

csharp

Copy code

if (Path.GetFullPath(filePath).StartsWith(\_baseFolder))

{

// Valid path

}

else

{

throw new UnauthorizedAccessException("Invalid file path.");

}

1. **Use Configuration for Base Folder**: Instead of hardcoding the wwwroot folder path, retrieve it from appsettings.json:

json

Copy code

"FileStorage": {

"BaseFolder": "wwwroot/assets/images"

}

Inject the configuration value in the FileRepository:

csharp

Copy code

\_baseFolder = configuration["FileStorage:BaseFolder"];

1. **Exception Handling**: Add proper exception handling when saving files to avoid crashes if disk space is full or the folder is inaccessible.

**Overall Feedback**

* **Architecture**: Clean and well-structured.
* **Scalability**: Easy to extend due to separation of concerns.
* **Suggestions**: Minor improvements to enhance error handling, logging, and configuration management.

You're on the right track! With these adjustments, your application will be even more robust and maintainable.

**Save CustomerReview and FileDetails together with one to one relationship.**

1. Create CustomerReview model and FileDetails model. For creating one to one model we need to have FileDetailsId as a (fk) and FileDetails property in the CustomerReview model.
2. Create an Angular form to get the the data from template in the component.
3. Add the model into the angular’s FormData(). First make CustomerReview model to string using JSON.stringify() and add to FormData() object using append() method and then add the file using append() method.
4. Prepare the Controller. The Controller can fetch the data from request. The extract the FormData and File Data and then process.
5. Make CustomerReview model exactly as the datatype which was defined for ef.