Nice — below is a complete, ready-to-use implementation for the Employee REST API you described. It includes the model, EF Core ApplicationDbContext, controller with all required endpoints (including both route-parameter and query-string department filters), and minimal Program.cs to run the app on port **8080** with Swagger enabled.

You can copy each file into your ASP.NET Core project (dotnetapp) and follow the commands you listed to restore, migrate and run.

**Files & code**

**1) Models/Employee.cs**

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace DotnetApp.Models

{

public class Employee

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

[Required]

public string Name { get; set; }

public string Department { get; set; }

[Column(TypeName = "decimal(18,2)")]

public decimal Salary { get; set; }

public string Position { get; set; }

}

}

**2) Data/ApplicationDbContext.cs**

using Microsoft.EntityFrameworkCore;

using DotnetApp.Models;

namespace DotnetApp.Data

{

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options)

{

}

public DbSet<Employee> Employees { get; set; }

}

}

**3) Controllers/EmployeesController.cs**

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using DotnetApp.Data;

using DotnetApp.Models;

namespace DotnetApp.Controllers

{

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

[ApiController]

public class EmployeesController : ControllerBase

{

private readonly ApplicationDbContext \_db;

public EmployeesController(ApplicationDbContext db)

{

\_db = db;

}

// GET: /api/employees

[HttpGet]

public async Task<ActionResult<IEnumerable<Employee>>> GetEmployees()

{

var list = await \_db.Employees.AsNoTracking().ToListAsync();

return Ok(list);

}

// GET: /api/employees/{id}

[HttpGet("{id:int}")]

public async Task<ActionResult<Employee>> GetEmployeeById(int id)

{

var emp = await \_db.Employees.FindAsync(id);

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

return Ok(emp);

}

// GET: /api/employees/filter?department=IT

[HttpGet("filter")]

public async Task<ActionResult<IEnumerable<Employee>>> GetEmployeesByDepartment([FromQuery] string department)

{

if (string.IsNullOrWhiteSpace(department))

{

// Return all if no department specified (or you can return BadRequest)

var all = await \_db.Employees.AsNoTracking().ToListAsync();

return Ok(all);

}

var filtered = await \_db.Employees

.AsNoTracking()

.Where(e => e.Department != null && e.Department.ToLower() == department.ToLower())

.ToListAsync();

return Ok(filtered);

}

// GET: /api/employees/department/{departmentName} (route parameter filter)

[HttpGet("department/{departmentName}")]

public async Task<ActionResult<IEnumerable<Employee>>> GetEmployeesByDepartmentRoute(string departmentName)

{

if (string.IsNullOrWhiteSpace(departmentName))

return BadRequest("departmentName cannot be empty.");

var filtered = await \_db.Employees

.AsNoTracking()

.Where(e => e.Department != null && e.Department.ToLower() == departmentName.ToLower())

.ToListAsync();

return Ok(filtered);

}

// POST: /api/employees

[HttpPost]

public async Task<ActionResult<Employee>> CreateEmployee([FromBody] Employee employee)

{

if (employee == null) return BadRequest();

// Server side validation examples:

if (string.IsNullOrWhiteSpace(employee.Name))

return BadRequest("Name is required.");

\_db.Employees.Add(employee);

await \_db.SaveChangesAsync();

// Returns 201 Created with Location header pointing to GET /api/employees/{id}

return CreatedAtAction(nameof(GetEmployeeById), new { id = employee.Id }, employee);

}

}

}

**4) Program.cs (minimal for .NET 6/7 style)**

using Microsoft.EntityFrameworkCore;

using DotnetApp.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

// Configure DB: use SQL Server if connection string provided, otherwise use InMemory for local testing.

var configuration = builder.Configuration;

var connString = configuration.GetConnectionString("DefaultConnection");

if (!string.IsNullOrEmpty(connString))

{

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

options.UseSqlServer(connString));

}

else

{

// fallback for simpler local execution (useful for tests that don't require a real DB)

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

options.UseInMemoryDatabase("InMemoryEmployeeDb"));

}

var app = builder.Build();

// Ensure DB created (InMemory or SQL Server - for development convenience)

using (var scope = app.Services.CreateScope())

{

var db = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

db.Database.EnsureCreated();

}

// Configure middleware

if (app.Environment.IsDevelopment() || true) // keep swagger enabled in test envs

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

// Force Kestrel to listen at port 8080 if you need to; you can also set launchSettings.json

app.Urls.Add("http://localhost:8080");

app.Run();

Note: If you prefer to set the URL via launchSettings.json or environment variables, remove app.Urls.Add(...) and configure accordingly.

**5) appsettings.json (add connection string if you will use SQL Server)**

{

"ConnectionStrings": {

"DefaultConnection": "User ID=sa;password=examlyMssql@123;server=localhost;Database=appdb;Trusted\_Connection=False;Persist Security Info=False;Encrypt=False"

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*"

}

**6) Properties/launchSettings.json (optional — ensure port 8080)**

{

"profiles": {

"DotnetApp": {

"commandName": "Project",

"dotnetRunMessages": true,

"launchBrowser": true,

"launchUrl": "swagger",

"applicationUrl": "http://localhost:8080;https://localhost:7243",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

}

}

**How to run (commands — same as your note)**

1. Open terminal in project root (where dotnetapp is located if that's the folder).
2. cd dotnetapp
3. dotnet restore
4. (Optional) If using EF migrations with SQL Server:
   * dotnet new tool-manifest (if not created)
   * dotnet tool install --local dotnet-ef --version 6.0.6
   * dotnet dotnet-ef migrations add InitialSetup
   * dotnet dotnet-ef database update
5. dotnet run — the app will listen on http://localhost:8080 (Swagger will be available).
6. Open Swagger UI: http://localhost:8080/swagger/index.html (or use the full deployment URL provided by your platform).

If using the sample SQL Server provided by your environment, follow the sqlcmd steps you included, create the database appdb and then run dotnet dotnet-ef database update.

**Endpoints summary (matches your spec)**

* GET /api/employees — returns 200 with list of employees.
* GET /api/employees/{id} — returns 200 with employee or 404 if not found.
* GET /api/employees/filter?department={departmentName} — returns 200 with employees filtered by query string.
* GET /api/employees/department/{departmentName} — returns 200 with employees filtered by route parameter.
* POST /api/employees — accepts JSON body, creates a new employee, returns **201 Created** with Location header pointing to GET /api/employees/{id}.

Example POST body:

{

"Name": "John Doe",

"Department": "IT",

"Salary": 75000.00,

"Position": "Software Engineer"

}

**Quick testing examples (curl)**

* Get all:

curl http://localhost:8080/api/employees

* Get by ID:

curl http://localhost:8080/api/employees/1

* Filter by query string:

curl "http://localhost:8080/api/employees/filter?department=IT"

* Filter by route:

curl http://localhost:8080/api/employees/department/IT

* Create:

curl -X POST http://localhost:8080/api/employees \

-H "Content-Type: application/json" \

-d '{"Name":"John Doe","Department":"IT","Salary":75000,"Position":"Software Engineer"}'