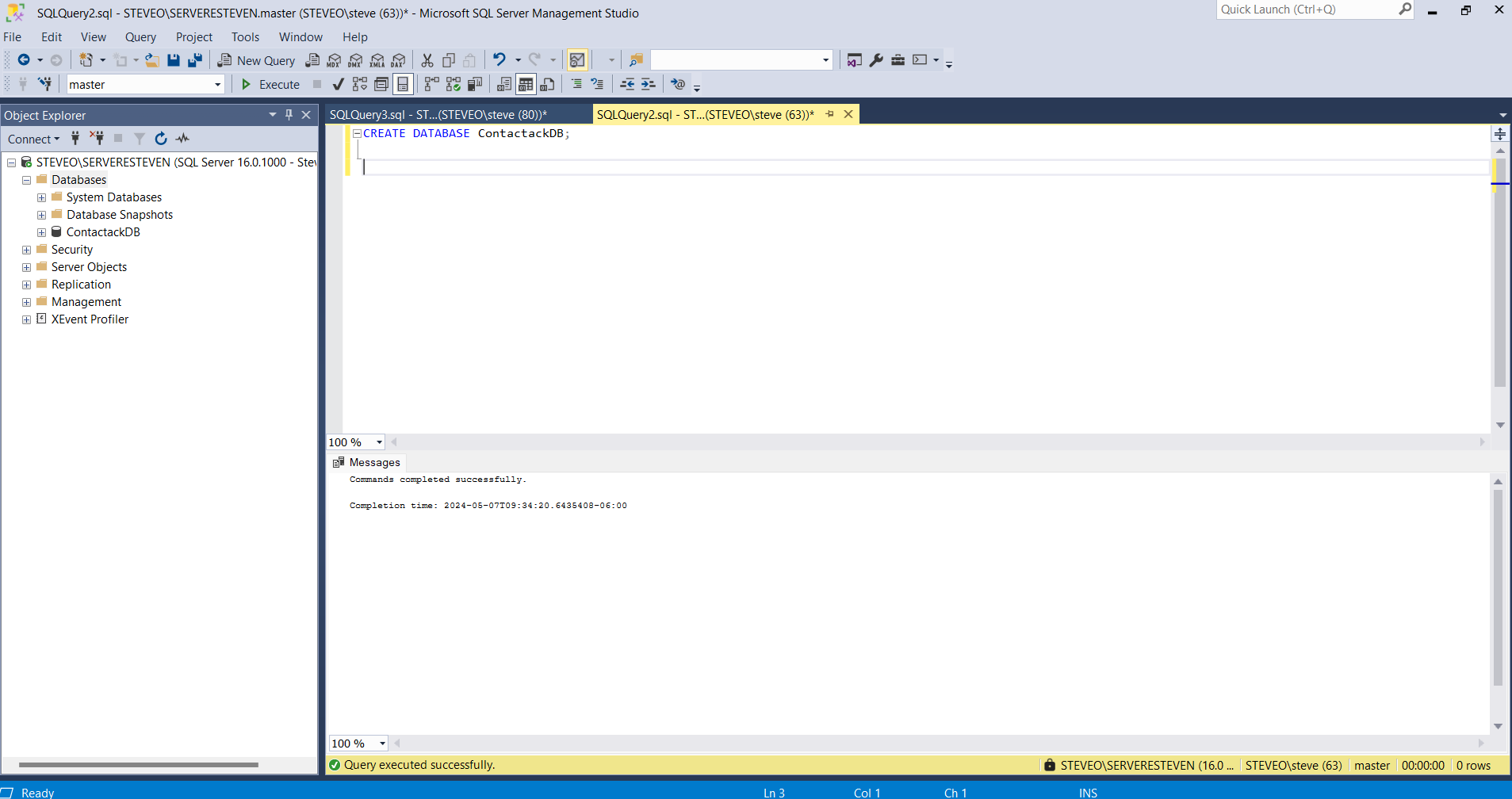
**Paso 1: Configuración del Entorno**

Antes de empezar, asegúrate de tener instalados los siguientes componentes:

* Visual Studio o Visual Studio Code
* .NET Core SDK (versión 3.1 o superior)
* Node.js (versión 14 o superior)
* SQL Server (local o en la nube)

**Paso 2: Creación de la Base de Datos**

CREATE DATABASE ContactackDB;



USE ContactackDB;

CREATE TABLE Clientes (

ClienteId INT PRIMARY KEY IDENTITY,

Nombre NVARCHAR(100),

Direccion NVARCHAR(200),

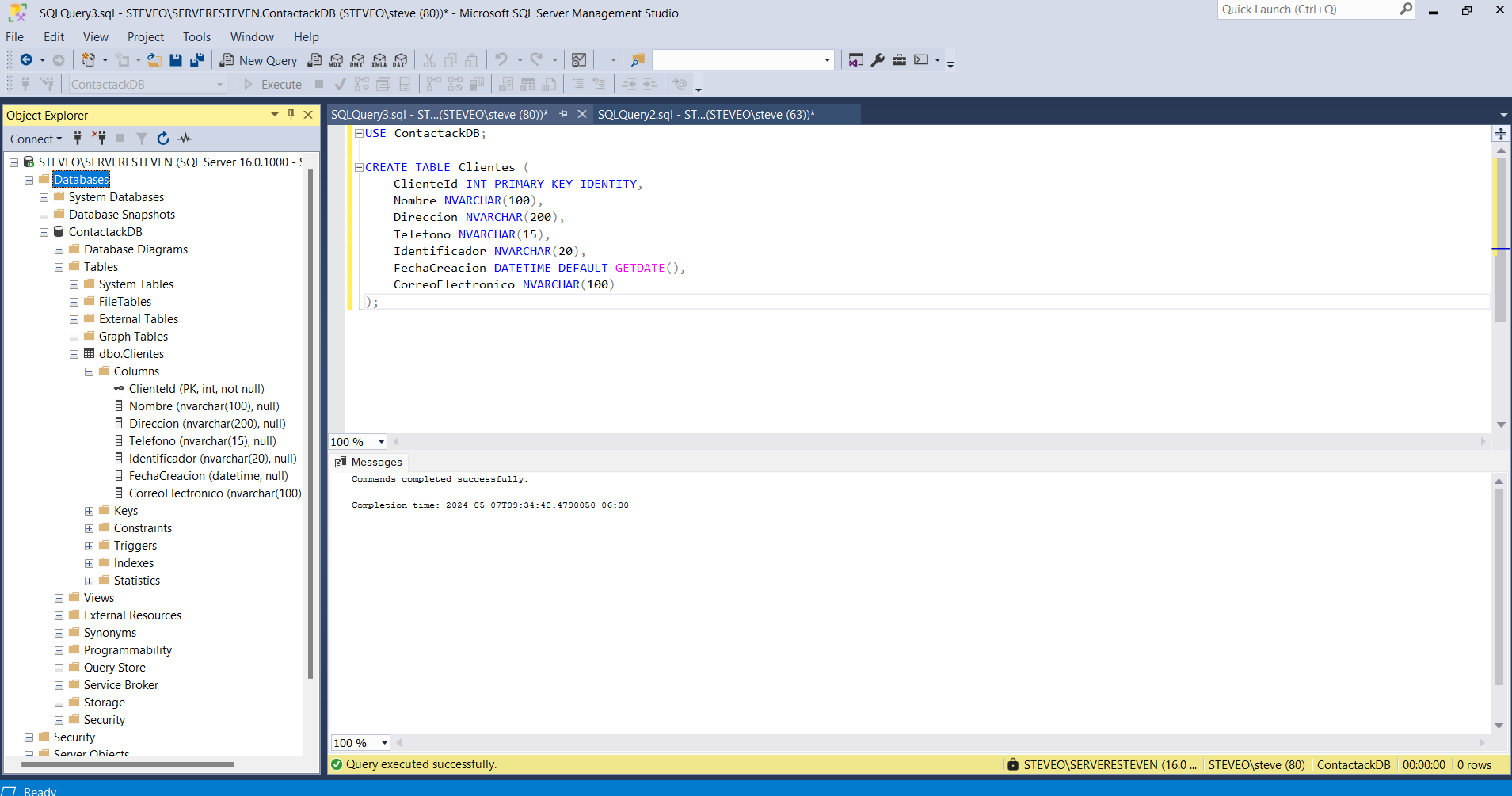
Telefono NVARCHAR(15),

Identificador NVARCHAR(20),

FechaCreacion DATETIME DEFAULT GETDATE(),

CorreoElectronico NVARCHAR(100)

);



**Paso 3: Creación del Backend en .NET Core**

1. Crea un proyecto en Visual Studio o Visual Studio Code utilizando el tipo de proyecto "ASP.NET Core Web API":

dotnet new webapi -n ClientAPI

cd ClientAPI

1. Instala los paquetes NuGet necesarios:

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer

1. Configura Entity Framework Core para conectar con la base de datos **ClientDB**:
   * Abre el archivo **Startup.cs** y configura el contexto de base de datos con una cadena de conexión adecuada:

C#

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ClientDbContext>(options =>

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

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = false,

ValidateAudience = false,

ValidateLifetime = true,

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKeyHere")),

ValidateIssuerSigningKey = true

};

});

services.AddControllers();

}

1. Crea un contexto de base de datos para Entity Framework Core:
   * Añade un archivo **ClientDbContext.cs**:

C#

public class ClientDbContext : DbContext

{

public DbSet<Cliente> Clientes { get; set; }

public ClientDbContext(DbContextOptions<ClientDbContext> options) : base(options) { }

}

1. Define el modelo **Cliente**:
   * Añade un archivo **Cliente.cs**:

C#

public class Cliente

{

public int ClienteId { get; set; }

public string Nombre { get; set; }

public string Direccion { get; set; }

public string Telefono { get; set; }

public string Identificador { get; set; }

public DateTime FechaCreacion { get; set; }

public string CorreoElectronico { get; set; }

}

1. Crea el controlador **ClientesController** para manejar el CRUD de la tabla **Clientes**:

C#

[ApiController]

[Route("[controller]")]

public class ClientesController : ControllerBase

{

private readonly ClientDbContext \_context;

public ClientesController(ClientDbContext context)

{

\_context = context;

}

[HttpGet]

public async Task<IActionResult> GetAll()

{

var clientes = await \_context.Clientes.ToListAsync();

return Ok(clientes);

}

[HttpGet("{id}")]

public async Task<IActionResult> GetById(int id)

{

var cliente = await \_context.Clientes.FindAsync(id);

if (cliente == null)

{

return NotFound();

}

return Ok(cliente);

}

[HttpPost]

public async Task<IActionResult> Create(Cliente cliente)

{

\_context.Clientes.Add(cliente);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetById), new { id = cliente.Id }, cliente);

}

[HttpPut("{id}")]

public async Task<IActionResult> Update(int id, Cliente cliente)

{

if (id != cliente.Id)

{

return BadRequest();

}

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

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!\_context.Clientes.Any(c => c.Id == id))

{

return NotFound();

}

else

{

throw;

}

}

return NoContent();

}

[HttpDelete("{id}")]

public async Task<IActionResult> Delete(int id)

{

var cliente = await \_context.Clientes.FindAsync(id);

if (cliente == null)

{

return NotFound();

}

\_context.Clientes.Remove(cliente);

await \_context.SaveChangesAsync();

return NoContent();

}

}

1. Crea el controlador **AuthController** para manejar JWT:

C#

[ApiController]

[Route("[controller]")]

public class AuthController : ControllerBase

{

[HttpGet("token")]

public IActionResult GetToken()

{

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKeyHere"));

var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(

issuer: null,

audience: null,

claims: null,

expires: DateTime.Now.AddMinutes(10),

signingCredentials: creds);

return Ok(new JwtSecurityTokenHandler().WriteToken(token));

}

[HttpGet("validate")]

public IActionResult ValidateToken()

{

return Ok("Token is valid");

}

}

)); } [HttpGet("validate")] public IActionResult ValidateToken() { return Ok("Token is valid"); } }

**Paso 4: Configuración del Frontend en ReactJS**

1. Crea un nuevo proyecto React:

npx create-react-app client-app

cd client-app

1. Instala los paquetes necesarios para interactuar con el backend y manejar JWT:

npm install axios jwt-decode

1. Crea una página para listar clientes:
   * Abre **src/App.js** y modifica para crear una lista de clientes:

Jsx

import React, { useEffect, useState } from 'react';

import axios from 'axios';

const App = () => {

const [clientes, setClientes] = useState([]);

useEffect(() => {

axios.get('http://localhost:5000/clientes')

.then((response) => {

setClientes(response.data);

})

.catch((error) => {

console.error(error);

});

}, []);

return (

<div>

<h1>Lista de Clientes</h1>

<ul>

{clientes.map((cliente) => (

<li key={cliente.Id}>

{cliente.Nombre} - {cliente.Direccion}

</li>

))}

</ul>

</div>

);

};

export default App;

1. Implementa Bootstrap para mejorar el estilo de la aplicación:
   * Instala Bootstrap:

npm install bootstrap

* + Importa Bootstrap en **src/index.js**:

Jsx

import 'bootstrap/dist/css/bootstrap.min.css';

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

1. Configura CORS en el backend para permitir solicitudes desde React:
   * Abre el archivo **Startup.cs** y añade la configuración para permitir CORS:

C#

public void ConfigureServices(IServiceCollection services)

{

services.AddCors(options =>

{

options.AddDefaultPolicy(builder =>

{

builder.WithOrigins("http://localhost:3000")

.AllowAnyHeader()

.AllowAnyMethod();

});

});

services.AddDbContext<ClientDbContext>(options =>

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

services.AddControllers();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseCors();

app.UseAuthentication();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

**Paso 5: Pruebas y Validación**

1. Inicia el backend de .NET Core:

dotnet run

1. Inicia el frontend de React:

npm start

1. Verifica que la lista de clientes se muestre correctamente en el navegador (en **http://localhost:3000/**).
2. Prueba los endpoints de autenticación:
   * Para obtener un token JWT, puedes utilizar **curl** o Postman:

curl http://localhost:5000/auth/token

* + Para validar el token JWT, utiliza el token obtenido anteriormente:

TOKEN="your-jwt-token" curl -H "Authorization: Bearer $TOKEN" http://localhost:5000/auth/validate

1. Para crear, actualizar, y eliminar clientes, puedes usar Postman o una herramienta similar para enviar solicitudes POST, PUT, y DELETE.