Technical Test for Junior Developers

Juan Diego Rojas Aguilar

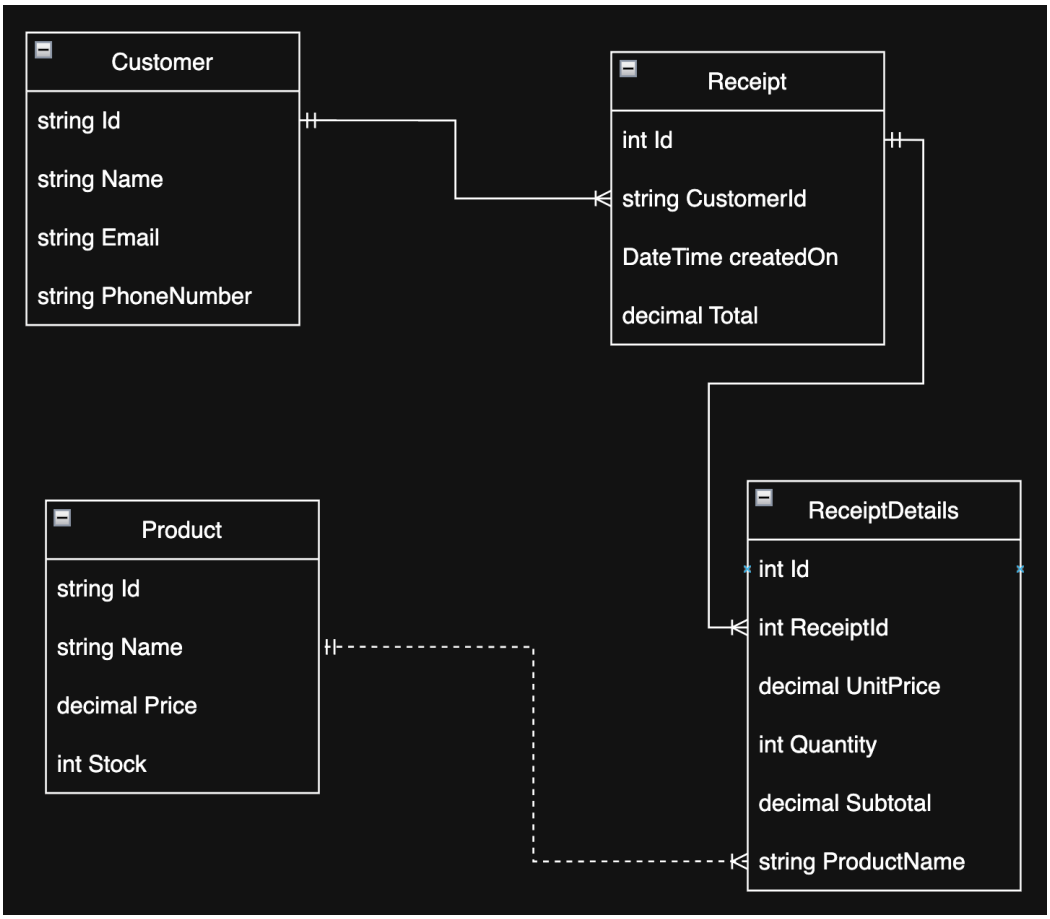
Para ConnexusIT

Table of Contents

Technical Test for Junior Developers

Sección 1: Desarrollo de aplicación

Esquema de la base de datos[1]



Scripts de ejecución:

- `docker compose up -d`
- `dotnet ef migrations add init`
- `dotnet ef database update`

Sección 3: Preguntas Teóricas

Preguntas:

1. Explique la diferencia entre REST y SOAP en el desarrollo de APIs.

2. Describa el concepto de la inyección de dependencias y su importancia en .NET
3. Explique el patrón MVC y cómo se aplica en .NET

Respuestas:

1. Cuando hablamos de REST y SOAP estamos hablando de dos maneras diferentes de representar la comunicación entre dos un sistema.

REST es usado en la comunicación por HTTP a través de JSON (principalmente) acepta diferentes tipos de seguridad como JWT o OAuth2, suele trabajarse de manera stateless para evitar el overhead en la lectura de datos, y es usado junto a los CRUD para hacer APIs.

SOAP por el otro lado es mayormente usado para comunicaciones entre sistemas y no es normalmente usado para representar información pues suele ser más pesado y demorado que el JSON. Tiene soporte ACID siendo excelente para generar transacciones entre sistemas.

Ejemplo de REST:

GET http://miapi.ejemplo.com/products/42

Resultado:

{“id”: 42, “nombre”: “Producto 1”, “precio”: 120.34}

Ejemplo SOAP:

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns1:GetCDLStat xmlns:ns1="http://cropscape.csiss.gmu.edu/CDLService/">
      <year>2011</year>
      <fips>19003,19029</fips>
      <format>CSV</format>
    </ns1:GetCDLStat>
  </soapenv:Body>
</soapenv:Envelope>
```

(a)

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns1:GetCDLStatResponse xmlns:ns1="http://cropscape.csiss.gmu.edu/CDLService/">
      <returnURL>
        http://nassgeodata.gmu.edu/nass_data_cache/CDL_2011_clip_352118264.csv
      </returnURL>
    </ns1:GetCDLStatResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

(b)

2. La inyección de dependencias es un patrón de diseño que nos permite fácilmente dividir las responsabilidades en diferentes archivos (normalmente basados en interfaces) lo cual nos permite desacoplar el funcionamiento de más bajo nivel con otra lógica, esto nos permite cambiar rápidamente el funcionamiento de nuestra aplicación .Net nos provee con herramientas en su builder que nos permite fácilmente adaptar interfaces a sus implementaciones asegurando un comportamiento *singleton* (builder.Services.AddScoped<Interface, InterfaceImplementation>).
3. El patron MVC (Modelo Vista Controlador) es el patrón, yo lo llamaría arquitectura, más común y usado por su simplicidad, efectividad y simplemente uso general que se le puede dar.

   Se trata de dividir nuestra aplicación en tres capas:

El Modelo que define las entidades en la base de datos, sus características y relaciones.

El Controlador que se encarga de procesar y enviar las solicitudes que vienen del navegador hacía la base de datos, procesos de parseo de datos, transformación de los Data Transfer Objects y el manejo de errores.

La Vista que se encarga de mostrar de manera humana al usuario los resultados de las consultas, esto puede ser desde mensajes de error o JSONs formateados hasta el frontend viéndolo desde una perspectiva mas general de la aplicación.

Sección 4: Depuración de código

Ejercicio 1:

```
public void CalcularTotal()
{
    decimal total = 0;
    foreach (var item in Items)
    {
        total += item.Cantidad * item.Precio
    }
    return total;
}
```

El error es el tipo de retorno, acá tenemos un void pero necesitamos usar un decimal, podríamos también aceptar Items como parámetro así la función puede ser reutilizada. Una opción mas arriesgada es castear decimal por cada item. Esto puede funcionar para asegurar un comportamiento pero puede llegar a dar problema con wrappers como IEnumerable.

Por otro lado podríamos validar que precio o cantidad sean positivos y no nulos.

Y faltaba un punto y coma al final del cálculo.

```
public decimal CalcularTotal(decimal[] items)
{
  decimal total = 0;

  foreach (var item in Items)
  {
    if (item.Cantidad > 0 && item.Precio > 0)
    {
      total += item.Cantidad * item.Precio;
    }
  }

  return total;
}
```

Ejercicio 2:

```
public bool EsMayorDeEdad(int edad)
{
    if (edad = 18)
    {
        return true;
    }
    return false;
}
```

El error es en la validación de la edad pues no valida si es mayor de edad sino que es igual a 18, si el usuario tuviese 30 años le diría que no es mayor de edad. A parte no valida datos negativos, nulos o vacíos.

Tambien podríamos optar por otro tipo de dato (unsigneds, short, byte) si se quiere ahorrar un poco de memoria pero podría dar problemas de compatibilidad en algunas bases de datos o librerías.

Por último podríamos definir el rango de mayoría de edad pues puede variar con clientes internacionales.

```
public bool EsMayorDeEdad(int edad, int mayoriaDeEdad)
{
  if (edad ≥ mayoriaDeEdad) // 18 en caso de Colombia
  {
    return true;
  }
  return false;
}
```

Ejercicio 3:

```
public void ProcesarPago(Cliente cliente, decimal monto)
{
    if (cliente == null)
    {
        throw new Exception("Cliente no encontrado");
    }

    cliente.Saldo -= monto;
}
```

Este error es muy común, estamos validando el cliente pero no validamos si el monto es suficiente, lo cual puede generar montos negativos en situaciones donde no lo requerimos. En una nota personal, también sería una buena práctica tener una excepción personalizada pues nos permite capturar esos errores de manera ordenada y diferenciar entre una BadRequest y un error 500.

```
public void ProcesarPago(Cliente cliente, decimal monto)
{
  if (monto == null)
  {
    throw new Exception("Monto Invalido");
  }

  if (cliente == null)
  {
    throw new ClienteNoEncontradoException();
  }

  if (cliente.Saldo < monto)
  {
    throw new SaldoInsuficienteException();
  }

  cliente.Saldo -= monto;
}
```

Ejercicio 4.

```
public decimal CalcularPromedio(List<int> valores)
{
    int suma = 0;
    foreach (var valor in valores)
    {
        suma += valor;
    }
    return suma / valores.Count;
}
```

Acá el error resulta al final, no validamos el caso de una lista vacía que nos daría una infame división por 0. También estamos usando un int para guardar nuestro resultado lo cual podría arrebatar los puntos de precisión.

```
public decimal CalcularPromedio(List<int> valores)
{
  if (valores.Count == 0)
  {
    return 0.0;
    // Throw new DivideByZeroException();
  }

  decimal suma = 0;

  foreach(int valor in valores)
  {
    suma += (decimal) valor;
  }

  return suma / valores.Count;
}
```

Ejercicio 5:

```
public bool EsPalindromo(string palabra)
{
    string reversa = "";
    for (int i = 0; i < palabra.Length; i++)
    {
        reversa += palabra[i];
    }
    return palabra == reversa;
}
```

Aca el problema es que estamos copiando la palabra directamente y siempre va a dar true, aparte de eso no hay checkeo de whitespace o null.

En este código usé un ciclo while, evite crear un string y la comparación va hasta la mitad pues se presupone que la otra mitad debe ser igual pero en orden descendente.

```
bool EsPalindromo(string palabra)
 {
    if (IsNullOrWhiteSpace(palabra))
    {
      throw new Exception("Por favor ingresa una palabra");
    }

    int i = 0;

    while (i < (palabra.Length >> 1))
    {
        if (palabra[i] != palabra[palabra.Length - i - 1])
        {
          return false;
        }
        ++i;
    }
    return true;
 }
```