# Apunte de Estudio - Curso: Aprende Programación Backend en C# .NET

## Sección 1 - Introducción General

### Objetivo del Curso

Este curso está orientado al **desarrollo de backend** utilizando **C# y .NET**, abordando:

* Conceptos generales de backend aplicables a cualquier lenguaje.
* Fundamentos de .NET y su evolución.
* Desarrollo de APIs REST con ASP.NET.
* Uso del formato JSON para comunicación entre sistemas.

### Perfil del Curso

* Apto tanto para programadores con experiencia como para principiantes.
* Finalidad: comprender y construir **endpoints escalables** que manejen reglas de negocio y recursos.

## .NET: Historia y Estructura

### ¿Qué es un marco de trabajo (framework)?

Es una estructura que define:

* Dónde poner el código.
* Cómo organizarlo.
* Cómo conectarlo entre componentes.

### Evolución de .NET

* **.NET Framework (2002)**: Privado. Competencia directa de Java y PHP.
* **.NET Core (2015)**: Código abierto y multiplataforma. Llega hasta la versión 3.
* **.NET 5+ (2020 en adelante)**: Unificación. Ya no se usa “Core” ni “Framework” en el nombre.
  + Ej.: .NET 5, .NET 6, .NET 7, .NET 8.

### Diferencia entre C# y .NET

* **C#**: Lenguaje de programación.
* **.NET**: Plataforma o framework donde corre C# (entre otros lenguajes).

## ¿Qué es el Backend?

### Definición

* Parte lógica del sistema, no tiene interfaz gráfica.
* Contiene y ejecuta las **reglas de negocio**.
* Atiende solicitudes desde frontend, apps móviles, hardware o videojuegos.

### Accesibilidad

* Está en un **servidor público** accesible mediante URL (IP o dominio).
* Expone **endpoints** accesibles vía **API**.

### Conexiones comunes al backend:

* Frontend (Angular, React, JavaScript, etc.).
* Dispositivos IoT (ej. Arduino, lentes inteligentes).
* Juegos online.

### Reglas de negocio

* Son las condiciones y lógicas específicas de una aplicación.
* Ej.: En Uber, que un auto esté disponible o no, o en MercadoLibre, listar y comprar productos.

## Qué es una API y sus componentes

### API (Application Programming Interface)

Es la interfaz que permite a otras aplicaciones comunicarse con el backend mediante **endpoints**.

### Endpoints

* Son rutas (ej.: /api/login, /api/products).
* Cada uno corresponde a una acción: iniciar sesión, obtener productos, etc.

## Protocolo HTTP

### ¿Qué es HTTP?

* “HyperText Transfer Protocol”: protocolo de transferencia de hipertexto.
* Permite enviar y recibir datos estructurados (como JSON).

### Verbos HTTP

* **GET**: Obtener datos.
* **POST**: Enviar nuevos datos.
* **PUT**: Actualizar datos.
* **DELETE**: Eliminar datos.
* **PATCH**: Actualización parcial (no siempre utilizado).

### Códigos de respuesta HTTP

* **1xx**: Informativos.
* **2xx**: Éxito (ej.: 200 OK).
* **3xx**: Redirecciones.
* **4xx**: Errores del cliente (ej.: 400 Bad Request).
* **5xx**: Errores del servidor (ej.: 500 Internal Server Error).

Los más comunes en el curso:

* **200**: Todo correcto.
* **400**: Datos enviados incorrectos.
* **500**: Error interno del servidor.

## Estructura de una Solicitud y una Respuesta HTTP

### Solicitud HTTP (Request)

1. **URL**: La ruta del recurso solicitado.
2. **Verbo (método)**: GET, POST, etc.
3. **Headers**: Información adicional (tipo de contenido, idioma, token de autorización).
4. **Body**: Solo en POST/PUT. Incluye datos en formato JSON, XML u otros.

Ejemplo de body JSON:

{  
 "usuario": "juan",  
 "password": "12345"  
}

### Respuesta HTTP (Response)

1. **Código de estado**: Indica el resultado (200, 400, 500, etc.).
2. **Headers**: Tipo de contenido (HTML, JSON, etc.), idioma, cookies, etc.
3. **Body**: Contenido devuelto, usualmente en formato JSON.

## Introducción al Formato JSON

### ¿Qué es JSON?

* “JavaScript Object Notation”.
* Formato de texto legible para intercambiar datos entre sistemas.
* No depende del lenguaje de programación.

### Ejemplo de Objeto JSON

{  
 "total": 100.50,  
 "customer": "Pedro",  
 "paid": true  
}

### Objetos Anidados y Colecciones

* Se pueden tener objetos dentro de otros objetos:

"address": {  
 "street": "Calle Juárez",  
 "number": 123,  
 "cp": 55235  
}

* Se pueden tener listas (arrays) de objetos:

"items": [  
 {  
 "description": "papas",  
 "amount": 1.0,  
 "quantity": 5  
 },  
 {  
 "description": "gaseosa",  
 "amount": 2.5,  
 "quantity": 2  
 }  
]

## Conclusión de la Sección

Con esta introducción, se sientan las bases para el desarrollo backend:

* Comprensión de .NET y C#.
* Uso de APIs y protocolos HTTP.
* Dominio de JSON como formato de intercambio de datos.

El conocimiento de estas bases es esencial para construir servicios backend robustos, reutilizables y escalables.

## Sección 2 - Introducción a C#

**Objetivo del curso**

* Curso intensivo de C# para personas que ya programan en otros lenguajes (Python, JavaScript, etc.) y necesitan familiarizarse con la sintaxis y conceptos básicos de C#.
* Es independiente del curso de backend principal.

**Configuración inicial**

1. **Entorno**: Visual Studio 2022.
2. **Proyecto**: Crear una aplicación de consola (ejemplo: *intro Csharp backend*).
3. **Framework**: .NET 8 (o versiones anteriores).
4. Ejecución: Compila y ejecuta el proyecto mostrando “Hello World”.

**Clases y objetos en C#**

* **Clase**: Es un molde para crear objetos.
* **Objeto**: Es una instancia concreta que tiene propiedades (datos) y comportamientos (métodos).

Ejemplos:

* Un auto tiene propiedades (color, puertas, llantas) y comportamientos (acelerar, frenar).

**Creación de clases**

* Se declara con class NombreClase { }.
* Las propiedades se definen usando:

public decimal Total { get; set; }

* + get: permite leer el valor.
  + set: permite modificar el valor.
  + Se puede quitar uno de los dos para restringir acceso. C# quita solo set;

**Convenciones**

* Las propiedades publicas empiezan con mayúscula.
* Las variables privadas comienzan con \_ (guion bajo).

| **Nivel de acceso** | **Convención** | **Ejemplo** |
| --- | --- | --- |
| private | camelCase o \_camelCase | edad, \_contador |
| public | PascalCase | Nombre, EdadTotal |

**Formas de crear objetos**

Tres maneras principales:

1. **Forma completa**:

Sale venta = new Sale();

1. **Usando var (azúcar sintáctica)**:

var venta = new Sale();

Nota: var en C# no es como en JavaScript; el tipo es fijo en compilación.

1. **Nueva forma simplificada**:

Sale venta = new();

**Constructores**

* Sirven para inicializar valores al crear un objeto.
* Se definen con el mismo nombre que la clase:

public Sale(decimal total) {

Total = total;

}

* this se usa para diferenciar variables de la clase y parámetros.

**Métodos (comportamiento)**

* Se definen dentro de la clase:

public string GetInfo() {

return "El total es " + Total;

}

* Se llaman desde el objeto creado.

**Encapsulamiento y modificadores de acceso**

* **public**: accesible desde cualquier lugar.
* **private**: accesible solo dentro de la clase.
* **protected**: accesible en la clase y en clases heredadas.

**Uso de objetos**

* Crear varios objetos a partir de la clase (molde).
* Llamar métodos y acceder propiedades según permisos.

Aquí tienes un **resumen claro y estructurado** de la clase que pasaste:

**1. Herencia (Inheritance)**

* **Concepto:** Permite **reutilizar el código** de una clase existente en otra clase.
* **Sintaxis:**

class Hija : Padre

* El constructor de la clase hija puede llamar al constructor de la clase padre usando : base(parametros).

**2. Acceso a miembros heredados**

* **Público (public):** Puede ser usado en cualquier lugar.
* **Privado (private):** Solo dentro de la clase padre, no se hereda.
* **Protegido (protected):** Se puede usar en la clase padre y también en las clases hijas, pero no fuera de ellas.

**3. Reutilización y ampliación**

* La clase hija hereda:
  + **Propiedades** y **métodos** públicos y protegidos.
* Puede:
  + Agregar nuevas propiedades (ej.: Tax).
  + Agregar nuevos métodos.
  + Llamar a los métodos heredados.

**4. Sobreescritura (Override)**

* Sirve para **cambiar el comportamiento de un método heredado**.
* Pasos:
  1. En el padre: marcar el método como virtual.
  2. En el hijo: usar override con el mismo nombre de método.

Ejemplo:

public virtual string GetInfo() { ... } // En el padre

public override string GetInfo() { ... } // En el hijo

* Cuando llamas al método desde un objeto de la clase hija, se ejecuta el método **sobrescrito**.

**5. Sobrecarga (Overload)**

* **Sobrecarga ≠ Sobreescritura**.
* Sobrecarga: **métodos en la misma clase con el mismo nombre pero diferentes parámetros** (tipo o cantidad).
* Ejemplo:

public string GetInfo() { ... }

public string GetInfo(string mensaje) { ... }

public string GetInfo(int numero) { ... }

**6. Ideas clave**

* **Herencia:** Reutiliza y extiende código.
* **Sobreescritura (override):** Cambia cómo funciona un método heredado.
* **Sobrecarga (overload):** Mismos nombres de método pero diferentes entradas de datos.

**¿Qué es una interfaz?**

- No se refiere a pantallas, sino a un concepto de programación.

- En POO, una interfaz es un contrato que establece reglas que una clase debe cumplir si decide implementarla.

- Es una forma de tipar y categorizar clases.

**¿Qué define una interfaz?**

- Puede contener propiedades (por ejemplo, decimal Total) y métodos (por ejemplo, void Save()).

- Las clases que implementan una interfaz deben implementar todo lo que ella define.

**¿Cómo se implementa?**

- Se usa ':' igual que con la herencia.

- Una clase puede implementar múltiples interfaces al mismo tiempo.

- Ejemplo:

public interface ISale { decimal Total { get; set; } }

public interface IShape { void Shape(); }

public class Venta : ISale, IShape {

public decimal Total { get; set; }

public void Shape() { /\* guardar en base de datos \*/ }

}

**Importancia de cumplir el contrato**

- Si una clase implementa una interfaz, debe cumplir todas sus reglas.

- El código no compila si falta algún método o propiedad.

**¿Para qué sirve?**

- Arquitectura limpia, principios SOLID y patrones de diseño se basan en interfaces.

- Permiten separar implementación de uso.

- Facilitan código más limpio, escalable y mantenible.

**Ejemplo práctico**

- Se puede tener una función que recibe cualquier objeto que implemente una interfaz específica.

- Ejemplo:

void Sum(IShape shape) { shape.Shape(); }

Sum(new Venta());

Sum(new Beer());

- Aunque Venta y Beer son distintas, ambas pueden ser procesadas gracias a la interfaz.

**Conclusión**

- Las interfaces permiten que el código sea abierto a extensiones pero cerrado a modificaciones.

- Se pueden agregar nuevas clases con diferentes comportamientos que cumplan el contrato.

- Son esenciales para modificar o escalar funciones sin romper el resto del sistema.

Aquí tenés un **resumen completo y claro** de la clase sobre **Generics en C#**, basado en la transcripción que me diste:

**Generics en C#**

**📌 ¿Qué son los Generics?**

Los **Generics** son una característica de la Programación Orientada a Objetos que permiten crear **clases y métodos reutilizables** para diferentes tipos de datos, sin perder seguridad de tipo.

* Se utilizan ampliamente en C#, Java y TypeScript.
* Son especialmente útiles en **colecciones** como List<T>, Dictionary<TKey, TValue>, etc.

**🧠 ¿Por qué usar Generics?**

* Para **evitar duplicar código** para distintos tipos de datos.
* Para crear estructuras que funcionen tanto con tipos primitivos (como int, string) como con clases personalizadas (objetos).
* Para mejorar la **reutilización**, la **claridad del código** y la **seguridad de tipos** en tiempo de compilación.

**🧱 Ejemplo explicado: Clase genérica MiLista<T>**

Se creó una clase MiLista<T> que:

* Usa T como tipo de dato genérico.
* Internamente tiene una List<T> privada.
* Tiene un **límite máximo** de elementos, definido en el constructor.
* Permite agregar elementos solo si no se supera ese límite.
* Tiene un método GetContent() que muestra los elementos almacenados.

MiLista<int> numbers = new MiLista<int>(5);

numbers.Add(1);

numbers.Add(2);

// ...

Console.WriteLine(numbers.GetContent());

También puede usarse con otros tipos:

MiLista<string> names = new MiLista<string>(5);

names.Add("Ana");

names.Add("Juan");

**🧪 Ejemplo con clases propias: MiLista<Cerveza>**

Se definió una clase Cerveza con:

* Propiedades Nombre y Precio.
* Un método sobrescrito ToString() para mostrar su contenido de forma legible.

public override string ToString()

{

return $"{Nombre} (${Precio})";

}

Luego se usó la clase genérica para almacenar cervezas:

MiLista<Cerveza> cervezas = new MiLista<Cerveza>(3);

cervezas.Add(new Cerveza("Erdinger", 5));

cervezas.Add(new Cerveza("Corona", 1));

cervezas.Add(new Cerveza("Delirium", 10));

cervezas.Add(new Cerveza("Paulaner", 6)); // No se agrega por superar el límite

**📌 Comportamiento importante**

* Si no se inicializa internamente la List<T>, dará error al hacer .Add(...).
* Se debe usar new List<T>() para crearla dentro del constructor.
* Gracias a la validación con Count < Limite, el método Add() evita superar el máximo permitido.

**🎯 Conclusión**

Los Generics permiten:

* **Crear estructuras flexibles** que funcionan con distintos tipos.
* **Reutilizar lógica** para enteros, textos u objetos personalizados.
* **Restringir comportamientos** como la cantidad máxima de elementos en una colección.
* **Sobrescribir ToString()** permite controlar cómo se muestran los objetos genéricos en pantalla.

# Serialización y Deserialización en C#

## ¿Qué es la serialización?

- Es el proceso de transformar un objeto en un formato que pueda almacenarse o transmitirse (ej. JSON).

- El formato JSON permite intercambiar información entre distintos lenguajes y tecnologías.

## ¿Qué es la deserialización?

- Es el proceso inverso: transformar un JSON (u otro formato) en un objeto en el lenguaje de programación.

## ¿Por qué se usa JSON?

- Es el formato elegido para el curso de backend.

- Es ampliamente usado en videojuegos, IoT, internet, etc.

## Ejemplo de serialización en C#

- Se define una clase con propiedades, por ejemplo, People con Nombre y Edad.

- Se crea un objeto y se usa `JsonSerializer.Serialize(objeto)` para convertirlo a JSON.

- Este JSON puede ser mostrado por consola o enviado a otros sistemas.

## Uso de métodos estáticos

- `JsonSerializer.Serialize` es un método estático, por lo tanto no requiere instanciar la clase.

- En general, los métodos o propiedades estáticas se usan directamente desde la clase.

## Syntactic sugar (azúcar sintáctica) en métodos

- Si un método tiene solo una línea, se puede escribir como: `=> expresión;` en lugar de usar llaves y return.

## Ejemplo de deserialización

- Se toma un string en formato JSON y se convierte en objeto con `JsonSerializer.Deserialize<Tipo>(json)`.

- Esto reconstruye el objeto original con sus propiedades.

## Manejo de posibles valores nulos

- Si el string JSON es nulo o inválido, el objeto resultante puede ser `null`.

- Se recomienda usar `?` en la declaración de variables para indicar que pueden ser nulas.

## Conclusión

- Serializar y deserializar es clave para trabajar con backend y muchas otras tecnologías.

- Aunque frameworks automatizan este proceso, entenderlo es fundamental para resolver errores y desarrollar soluciones robustas.

**Expresiones Lambda en C#**

**Definición:**  
Son funciones anónimas (sin nombre) que pueden definirse directamente en el lugar donde se usan, sin necesidad de declararlas previamente. Se utilizan comúnmente como parámetros de **funciones de orden superior**.

**Sintaxis básica:**

(parametros) => expresión

**Características principales:**

* **Funciones anónimas**: no requieren un nombre ni declaración aparte.
* **Uso puntual**: ideales para código que se ejecuta una sola vez.
* **Inferencia de tipos**: si el contexto ya define los tipos, no es necesario escribirlos.
* **Parámetro único**: si hay solo uno, se pueden omitir paréntesis.
* **Bloques múltiples**: si la lógica requiere más de una línea, se usan llaves { } y return.

**Ejemplos:**

// Con dos parámetros

(int a, int b) => a - b;

// Con inferencia de tipos

(a, b) => a + b;

// Un solo parámetro, sin paréntesis

a => a \* 2;

// Varias líneas de código

a => {

a += 1;

return a \* 5;

};

Uso con funciones de orden superior:

int Sum(Func<int, int, int> fn, int numero)

{

return fn(numero, numero);

}

// Llamada con lambda como parámetro

var resultado = Sum((a, b) => a + b, 5); // resultado = 10

**Beneficios:**

* Menos código repetitivo.
* Más legibilidad en funciones simples.
* Útiles para pasar lógica directamente como parámetro.
* Combinan bien con LINQ y otras funciones de C#.

Perfecto, ya tengo tu transcripción y el documento de apuntes.  
Te armé un **resumen de LINQ** que sigue el mismo estilo y nivel de detalle que tienen las otras secciones de tu apunte, de modo que encaje de forma natural:

**LINQ en C#**

**Definición**  
LINQ (Language Integrated Query) es una extensión del lenguaje C# que permite trabajar con colecciones de datos de manera **declarativa**, usando una sintaxis similar a SQL.  
Su principal ventaja es poder **filtrar, ordenar, agrupar y seleccionar** elementos de colecciones (listas, arrays, datos de bases, XML, JSON, etc.) sin necesidad de escribir bucles manuales.

**Partes principales de LINQ**

1. **Origen de datos**  
   Puede ser cualquier colección o fuente compatible: listas, arrays, resultados de una base de datos, XML, JSON, etc.
2. **Consulta**  
   Es la instrucción que define cómo queremos transformar o filtrar los datos.  
   Puede escribirse en:
   * **Sintaxis de consulta**: parecida a SQL (from ... where ... orderby ... select ...).
   * **Sintaxis de funciones**: usando métodos de extensión (Where(...), OrderBy(...), Select(...)).
3. **Ejecución**
   * **Diferida**: la consulta no se ejecuta hasta recorrer el resultado (por ejemplo, en un foreach).
   * **Inmediata**: usando métodos como .ToList() o .ToArray(), se materializa el resultado en memoria.

**Ejemplo con sintaxis de consulta**

var names = new List<string>() { "Juan", "Pepe", "Ana", "Hugo", "Nico" };

var namesResult = from n in names

where n.Length > 3 && n.Length < 5

orderby n

select n;

foreach (var name in namesResult) // ejecución diferida

{

Console.WriteLine(name);

}

**Ejemplo equivalente con sintaxis de funciones**

var namesResult2 = names

.Where(n => n.Length > 3 && n.Length < 5) // filtra por longitud

.OrderByDescending(n => n) // orden descendente

.Select(n => n); // selecciona el mismo elemento

**Operaciones comunes en LINQ**

* **Where(...)** → Filtra elementos según una condición.
* **OrderBy(...)** / **OrderByDescending(...)** → Ordena elementos ascendente o descendente.
* **Select(...)** → Proyecta datos (selecciona campos o transforma elementos).
* **GroupBy(...)** → Agrupa elementos por una clave.
* **First() / FirstOrDefault()** → Devuelve el primer elemento (o valor por defecto si no hay resultados).
* **Any() / All()** → Verifica si algún elemento o todos cumplen una condición.
* **Count() / Sum() / Max() / Min() / Average()** → Operaciones agregadas.

**Consideraciones importantes**

* **No modifica la colección original**, siempre devuelve una nueva secuencia con el resultado.
* Permite combinar múltiples condiciones (&&, ||).
* El compilador traduce la sintaxis de consulta a la sintaxis de funciones antes de ejecutarla.
* Usar .ToList() en colecciones grandes puede impactar en el rendimiento porque fuerza la ejecución inmediata y copia todos los elementos a memoria.

📌 **Conclusión**  
LINQ unifica la forma de consultar datos en C#, permitiendo un código más limpio, legible y similar a SQL, con la ventaja de trabajar directamente sobre objetos y colecciones del lenguaje.