



**APLICACIÓN HÍBRIDA,
MONGODB Y C#**

Realizado por: Juan Luis Pérez Barrera

José Ruiz Alba

GRUPO 32

Índice de contenidos

1	Resumen	3
2	Objetivo	3
3	Descripción	3
4	¿Qué es MongoDB?	3
5	¿Qué es C#?	6
6	Herramientas	7
7	Estructura del Proyecto	7
8	Manual de uso	7
9	Implementación.....	8
9.1	Paso 1: Creación del proyecto en Visual Studio y instalación de los drivers para MongoDB ...	8
9.2	Paso 2. Administrar conexión y operaciones con la Base de Datos	15
9.2.1	Conectar la Base de Datos	15
9.2.2	Insertar elementos de la Base de Datos.....	16
9.2.3	Mostrar un elemento de la Base de Datos.....	16
9.2.4	Editar elementos de la Base de Datos.....	17
9.2.5	Borrar elementos de la Base de Datos	17
9.2.6	Listar elementos de la Base de Datos	18
9.2.7	Búsqueda de elementos en la Base de Datos	18
9.3	Paso 3. Modelo de Datos a Insertar	19
9.4	Paso 4. Estilo	23
10	Problemas	40
11	Mejoras.....	41
12	Conclusiones	41
13	Anexo	42
13.1	Links de interés	43
13.2	Instalar Visual Studio	43
13.3	Instalar MongoDB	44
13.4	Código de la aplicación	45

Tabla de Ilustraciones

Ilustración 1. Objeto en formato BSON	4
Ilustración 2. Ejecutar un programa con Visual Studio	7
Ilustración 3. Crear Proyecto en Visual Studio	8
Ilustración 4. Aplicación de consola (.NET Core)	9
Ilustración 5. Configuración de Proyecto	10
Ilustración 6. Proyecto Inicio	11
Ilustración 7. Instalación de Paquete NuGet	12
Ilustración 8. MongoDB.Driver	13
Ilustración 9. Proyecto Inicial con Drivers Instalados	14
Ilustración 10. Comprobación Instalación Driver	14
Ilustración 11. Listado de Series en consola	31
Ilustración 12. Visual Studio intalación	43
Ilustración 13. MongoDB instalación	44

1 Resumen

En el siguiente documento se pretende diseñar una pequeña aplicación sencilla de consola con las tecnologías de C# y MongoDB. En el siguiente documento se explicitarán las pautas para la instalación de ambas tecnologías, así como la instalación de entornos para facilitarnos el uso de las mismas.

2 Objetivo

La presente aplicación tiene como objetivo realizar una introducción sencilla a la tecnología de C# como al propio MongoDB, realizar una pequeña introducción hacia los driver de C# y MongoDB, junto a esto también intentamos indagar en las diferentes posibilidades que nos ofrece las tecnologías como son el plan MongoDB atlas, framework para desarrollar aplicaciones ASP .NET, etc.

3 Descripción

La aplicación desarrollada consiste en una aplicación de consola destinada a el manejo de series, usando una base de datos en local, del NoSQL, llamada MongoDB. La aplicación se caracteriza por estar enfocada a integrar Mongo con C#, para aprovechar la potencia de las tecnologías con respecto a su poder de escalabilidad y versatilidad, frente a otras.

4 ¿Qué es MongoDB?

MongoDB es una de las bases de datos más populares del momento, esta se caracteriza por ser un proyecto Open Source que forma parte de las bases de datos NoSQL (Not only SQL). MongoDB es una base de datos de documentos que ofrece una gran escalabilidad y flexibilidad, y un modelo de consultas e indexación avanzado.

MongoDB en lugar de almacenar datos en tablas como las bases de datos SQL, almacena datos en documentos flexibles similares a JSON, por lo que los campos pueden variar entre documentos y la estructura de datos puede cambiarse con el tiempo.

Estos documentos en formato Bson, se agrupan en colecciones, que son las estructuras de datos de MongoDB que se encarga de agrupar los diferentes elementos Bson.

El formato de los documentos es el siguiente:



Ilustración 1. Objeto en formato BSON

Este documento, puede asociarse a cualquier objeto que tengo en nuestra aplicación, es decir, la propia base de datos se encarga de guardar los objetos que nosotros creamos en nuestro código siguiendo este formato a través de los diferentes drivers, haciendo el proceso de transformar el objeto programado a este formato completamente transparente para el programador.

También, MongoDB es una base de datos distribuida en su núcleo, por lo que la alta disponibilidad, la escalabilidad horizontal y la distribución geográfica están integradas y son fáciles de usar con un plan gratuito.

A esto se suma en plan MongoDB Atlas que nos permite disponer de la una base de datos en la nube gratuitamente sin tener que contratar un previo servicio, solo con registrarnos en la web de MongoDB Atlas, dispondremos de una base de datos no relacional desplegada en la nube, con unos simples clicks de ratón.

En la actualidad, empresas como Bosch, Expedia, Forbes usan MongoDB, gracias a su potencial.

Algunas de las características principales de MongoDB son :

- **Consultas ad hoc** . Con MongoDb podemos realizar todo tipo de consultas. Podemos hacer búsqueda por campos, consultas de rangos y expresiones regulares. Además, estas consultas pueden devolver un campo específico del documento, pero también puede ser una función JavaScript definida por el usuario.
- **Indexación** . El concepto de índices en MongoDB es similar al empleado en bases de datos relacionales, con la diferencia de que cualquier campo documentado puede ser indexado y añadir múltiples índices secundarios.
- **Replicación** . Del mismo modo, la replicación es un proceso básico en la gestión de bases de datos. MongoDB soporta el tipo de replicación primario-secundario. De este modo, mientras podemos realizar consultas con el primario, el secundario actúa como réplica de datos en solo lectura a modo copia de seguridad con la particularidad de que los nodos secundarios tienen la habilidad de poder elegir un nuevo primario en caso de que el primario actual deje de responder.
- **Balanceo de carga** . Resulta muy interesante cómo MongoDB puede escalar la carga de trabajo. MongoDB tiene la capacidad de ejecutarse de manera simultánea en múltiples servidores, ofreciendo un balanceo de carga o servicio de replicación de datos, de modo que podemos mantener el sistema funcionando en caso de un fallo del hardware.
- **Almacenamiento de archivos** . Aprovechando la capacidad de MongoDB para el balanceo de carga y la replicación de datos, Mongo puede ser utilizado también como un sistema de archivos. Esta funcionalidad, llamada GridFS e incluida en la distribución oficial, permite manipular archivos y contenido.
- **Ejecución de JavaScript del lado del servidor**. MongoDB tiene la capacidad de realizar consultas utilizando JavaScript, haciendo que estas sean enviadas directamente a la base de datos para ser ejecutadas.

Para más información puedes visitar los siguientes links:

- <https://docs.mongodb.com/manual/crud/>

- <https://www.mongodb.com/cloud/atlas>
- <https://docs.mongodb.com/manual/tutorial/getting-started/>

5 ¿Qué es C#?

“C# es un lenguaje elegante, con seguridad de tipos y orientado a objetos que permite a los desarrolladores crear una gran variedad de aplicaciones seguras y sólidas que se ejecutan en .NET Framework. Puede usar C# para crear aplicaciones cliente de Windows, servicios web XML, componentes distribuidos, aplicaciones cliente-servidor, aplicaciones de base de datos entre otras”, según la empresa desarrolladora, en este caso Microsoft.

La sintaxis de C# es muy expresiva, pero también sencilla y fácil de aprender. Cualquier persona familiarizada con C, C++ o Java, reconocerá al instante la sintaxis de llaves de C#. Los desarrolladores que conocen cualquiera de estos lenguajes pueden empezar normalmente a trabajar en C# de forma productiva en un espacio breve de tiempo.

Algunas de las características principales de C# son :

- **Sencillez:** En comparación a los otros lenguajes antecesores de este, C# elimina cierto objetos y atributos innecesarios para que la acción de programar sea más intuitiva.
- **Modernidad:** Enfocado para dar solución a los temas actuales, también el lenguaje C# realiza de manera automática e intuitiva la incorporación de algunos objetos que con el paso de los años han sido necesarios a la hora de programar.
- **Seguridad:** Desde unas instrucciones para realizar acciones seguras y un mecanismo muy fuerte para la seguridad de los objetos.
- **Sistemas de tipos unificados:** Todos los datos que se obtienen al programar el lenguaje C# quedan guardadas en una base para que puedan ser utilizada posteriormente.
- **Extensibilidad:** Esta característica es muy positiva, debido a que puedes añadir tipos de datos básicos, operadores y modificadores a la hora de programar.
- **Versionable:** Dispone la característica de tener versiones, es decir, actualizarse y mejorar constantemente.
- **Compatible:** Tanto con sus antecesores como con Java y muchos otros lenguajes de programación, #C integra a todos estos para facilidad del programador.

Para más información puede visitar los siguientes links:

- <https://docs.microsoft.com/es-es/dotnet/csharp/getting-started/>
- https://www.w3schools.com/cs/cs_getstarted.asp

6 Herramientas

Para la realización de la aplicación se requiere una previa instalación de herramientas como Visual Studio, herramienta recomendada por el propio Microsoft para trabajar con la tecnología C# y como base de datos para guardar los datos utilizaremos MongoDB, en concreto utilizaremos MongoDB, como base de datos, como servidor en local, si el usuario decide desplegar la aplicación en la nube se pueden investigar el plan Atlas que nos permite disfrutar de una base de datos MongoDB desplegada en la nube de forma gratuita.

7 Estructura del Proyecto

La aplicación desarrollada cuenta con una estructura de 6 archivos con extensión .cs, en la que observamos:

- Actor.cs y Serie.cs, corresponden con los objetos que hemos definido
- MongoConnect.cs, se especifican las conexiones y operaciones con la base de datos
- Dialogo.cs, se establece el dialogo entre la aplicación y el usuario
- EstiloConsola.cs, la clase se encarga de imprimir los resultados por pantalla en formato tabla en consola
- Program.cs, clase principal donde se aloja el programa

8 Manual de uso

En este apartado se explicitan las diferentes pautas y requisitos necesarios para ejecutar la aplicación, para ejecutar nuestra aplicación es necesario tener previamente instalado en el equipo Visual Studio además de MongoDB en su ordenador en local, una vez instalado estas herramientas, en el caso que no esté instaladas la siguientes herramientas, podemos ir al Anexo de este documento donde encontraremos como instalar las diferentes herramientas podremos ejecutar la aplicación realizando los siguientes pasos los siguientes pasos:

1. Abrimos Visual Studio
2. Elegimos la opción Abrir un proyecto o una solución
3. Seleccionamos el archivo con extensión .sln, con el nombre CBD-PROYECTO.sln
4. Ejecutamos la aplicación, pulsando en el botón que parece en un recuadro en rojo

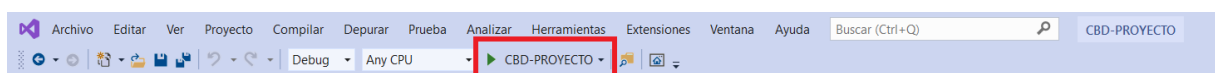


Ilustración 2. Ejecutar un programa con Visual Studio

Una vez ejecutada la aplicación nos aparecerá un menú principal con las diferentes funcionalidades de nuestra aplicación, si el usuario quiere ver cómo funciona cada opción que nos brinda la aplicación

puede visitar nuestra tutorial de YouTube en,
<https://www.youtube.com/playlist?list=PLIIMS2xXqAXmTC2nuxajMjShNXtgQILn1>

9 Implementación

En este apartado vamos a comentar los diferentes pasos que hemos seguido para desarrollar la aplicación. Hemos dividido el desarrollo en 4 pasos principales en los que en:

- Paso 1, Creamos un nuevo proyecto en Visual Studio e instalamos los drivers con MongoDB
- Paso 2, Conectamos nuestra nueva aplicación con la base de datos
- Paso 3, Realizamos un modelo de datos, en nuestro caso de Series
- Paso 4, Creamos unas clase auxiliares, para mejorar el aspecto de la aplicación

9.1 Paso 1: Creación del proyecto en Visual Studio y instalación de los drivers para MongoDB

En este apartado describiremos como crear nuestro primer proyecto de Visual Studio

1. Iniciamos la Herramienta Visual Studio, nos aparecerá la siguiente ventana:

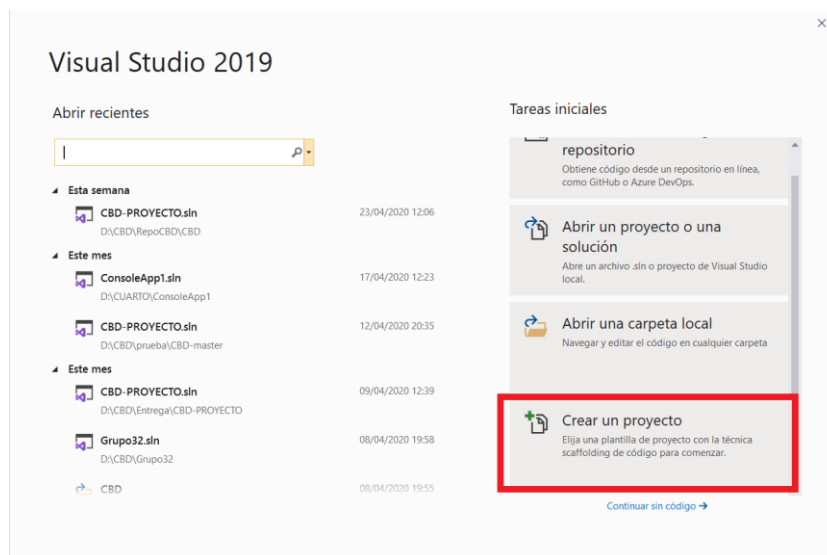


Ilustración 3. Crear Proyecto en Visual Studio

2. Elegimos la opción Crear un nuevo proyecto

En esta ventana nos aparece a la izquierda aquellos proyectos, que hemos abierto recientemente, en nuestro caso si acabamos de instalar la aplicación debería de NO aparecernos nada. A la derecha nos muestra un menú en el que podemos elegir las diferentes opciones que disponemos para poder abrir proyectos, en concreto nosotros queremos realizar uno nuevo, por lo que hacemos click en la opción “Crear un Proyecto”.

3. Elegimos la opción Aplicación de consola (.NET Core)

Una vez que creamos un proyecto se nos abre un menú de plantillas entre las que podemos elegir plantillas de para aplicaciones web siguiendo el modelo MVC, Aplicaciones Razor, Aplicaciones de consola.NET Core, entre otras, en concreto nosotros elegimos la opción marcada en rojo “Aplicación de consola (.NET Core)”

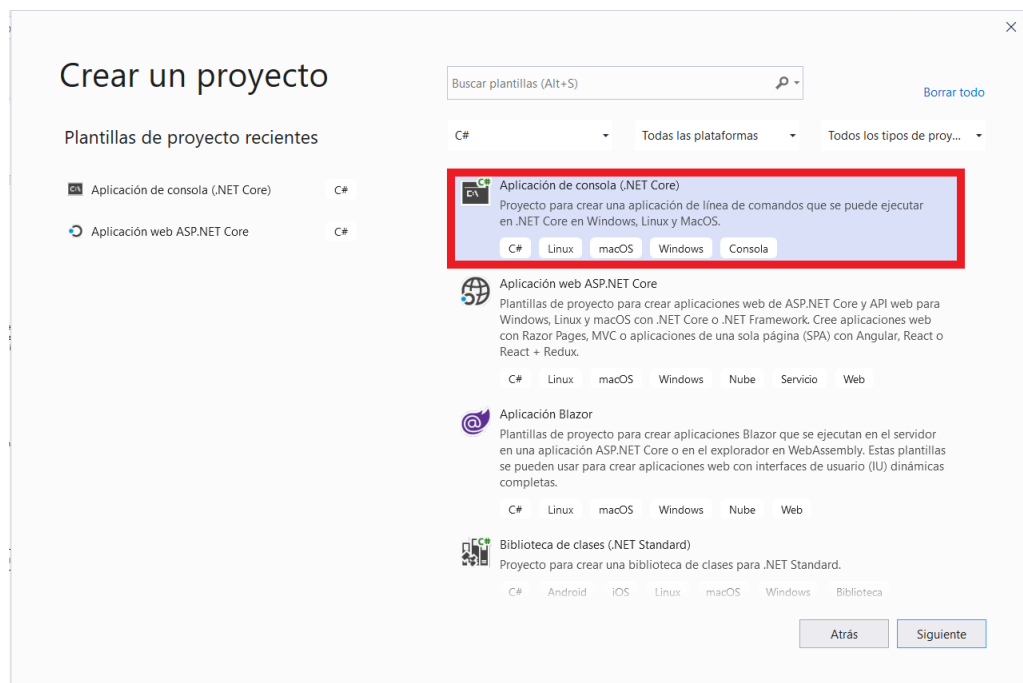
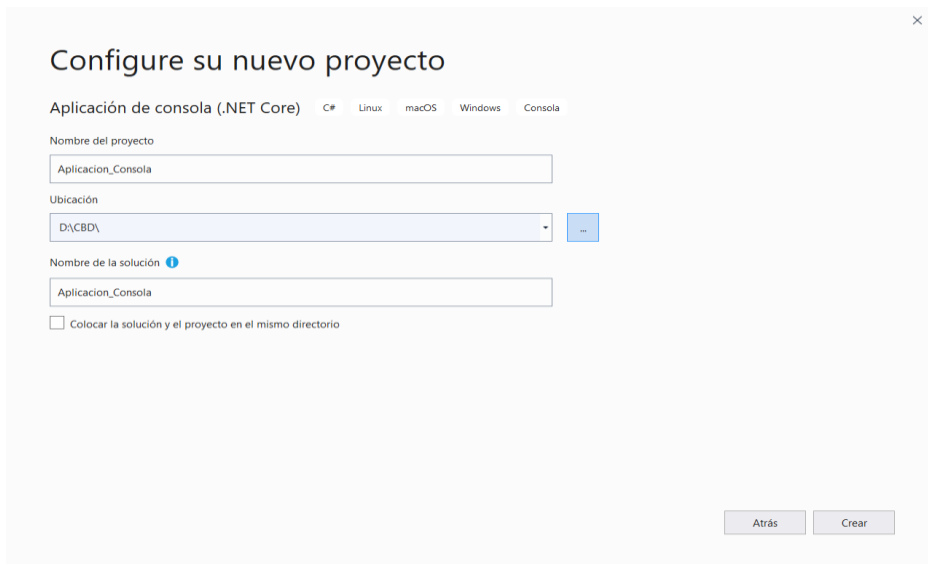


Ilustración 4. Aplicación de consola (.NET Core)

Tras elegir la plantilla que vamos a usar para elaborar el proyecto el siguiente paso consiste en establecer un nombre del proyecto y una ubicación donde se van a guardar los datos referentes al proyecto, una vez establecidos esos nombres, pulsamos en botón de crear.



Configure su nuevo proyecto

Aplicación de consola (.NET Core) C# Linux macOS Windows Consola

Nombre del proyecto

Aplicacion_Console

Ubicación

D:\CBD\

Nombre de la solución ⓘ

Aplicacion_Console

☐ Colocar la solución y el proyecto en el mismo directorio

Atrás Crear

Ilustración 5. Configuración de Proyecto

Una vez que hemos creado el proyecto, se nos abrirá un archivo Program.cs, el cual es nuestro archivo principal, el cual contiene el método main el cual nos permite ejecutar lo que hay dentro de ese main, y poder pintar los resultados por consola, en este caso si ejecutamos en método main, nada más crear el proyecto nos aparecerá por consola un Hello World.

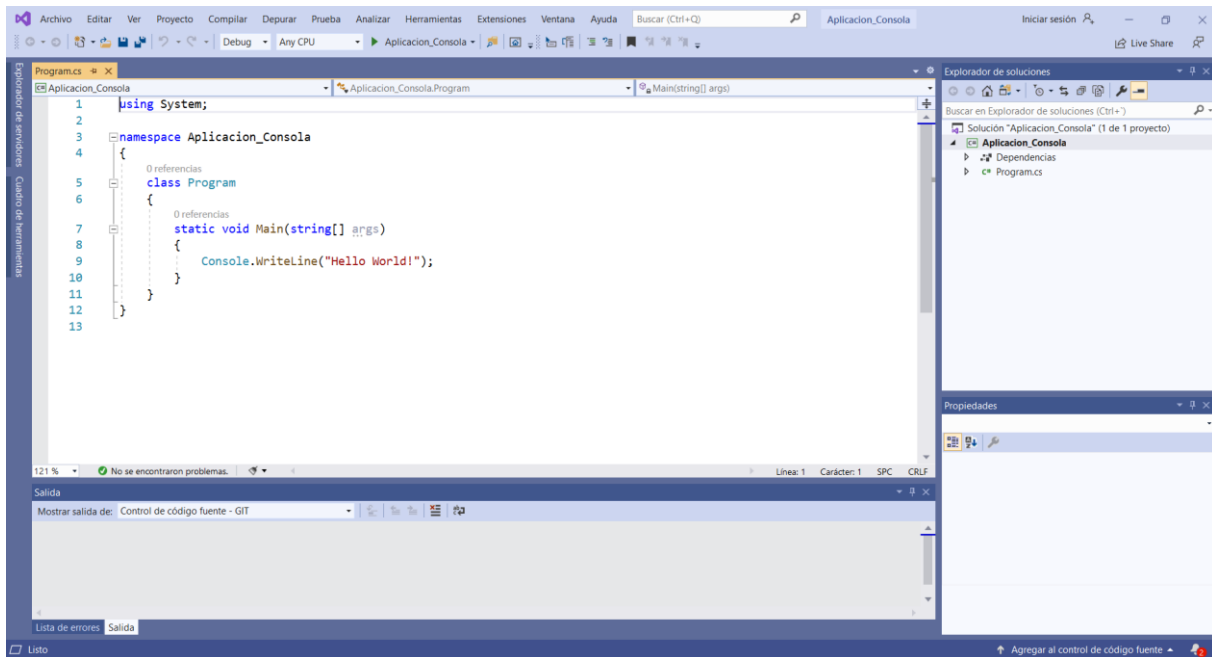


Ilustración 6. Proyecto Inicio

Ahora que tenemos creado el proyecto en Visual Studio y tenemos todo preparado para comenzar a conectar nuestra aplicación con la base de datos, el primer paso que tenemos que llevar a cabo es instalar los driver de mongoDB, en C#, para ello buscamos en el menú superior el apartado de herramientas, hacemos click en él, y dentro del desplegable, nos vamos a la OPCIÓN Administrador de paquetes NuGet, aquí aparecen varias opciones de como instalar un paquete NuGet, nosotros hemos elegido la opción Administrar paquetes NuGet para la solución.

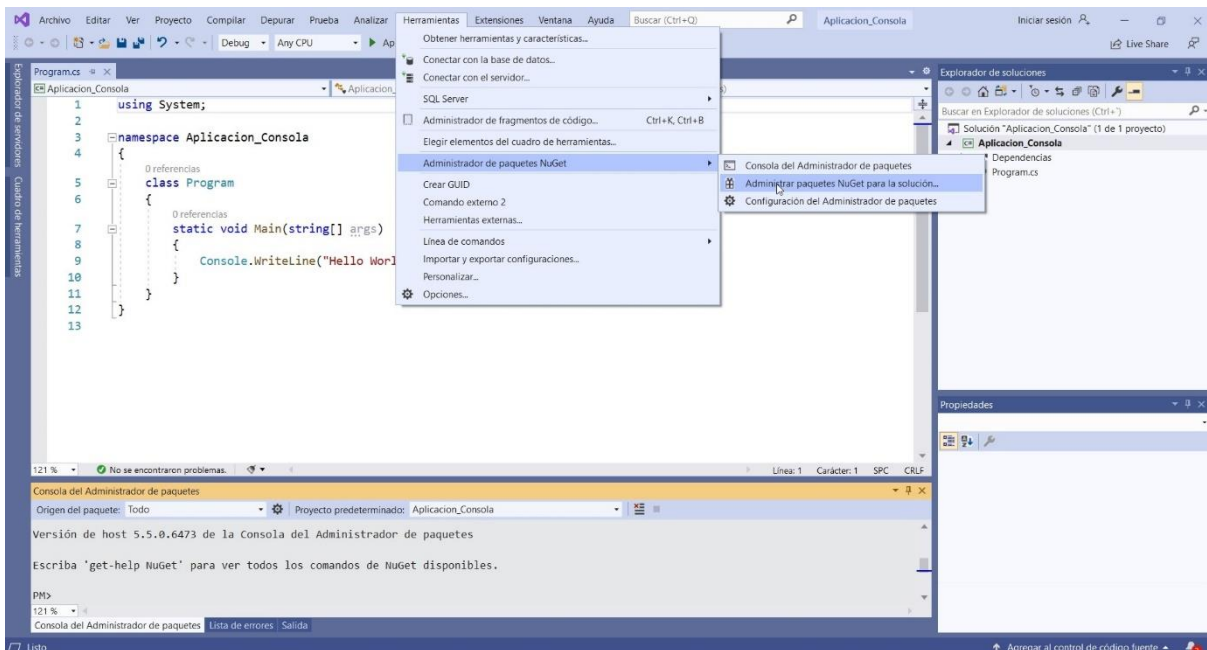


Ilustración 7. Instalación de Paquete NuGet

Una vez que elegimos la opción Administrar paquetes NuGet para la solución, se nos abre una pequeña interfaz gráfica, una vez que estemos en una interfaz gráfica como la aparece a continuación, hacemos click en la opción examinar, y escribimos el buscador mongo, y nos aparecerán opciones como las que vemos a continuación, nosotros tendremos que elegir, MongoDB.Driver, y procedes a su instalación

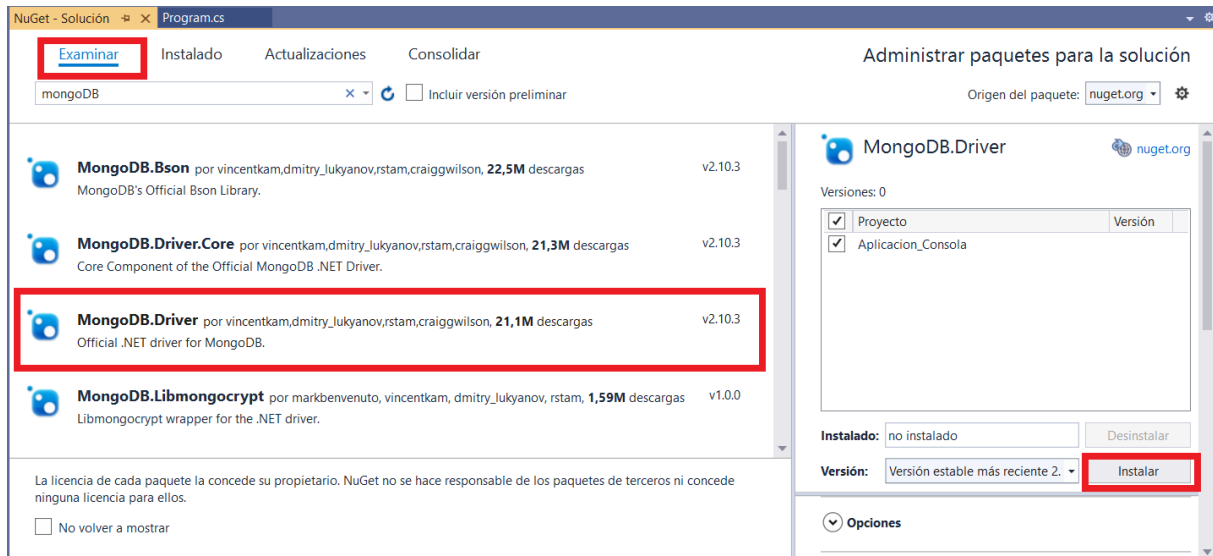


Ilustración 8. MongoDB.Driver

Una vez que pinchamos en la opción instalar nos aparecen un pequeño menú de en el que se nos muestran los objetos que se van a instalar al decidir instalar ese paquete NuGet en concreto, para continuar, pinchamos en la opción aceptar. Ya tras este paso tendríamos instalados todo lo referente al driver de MongoDB en nuestro proyecto listo para utilizarse.

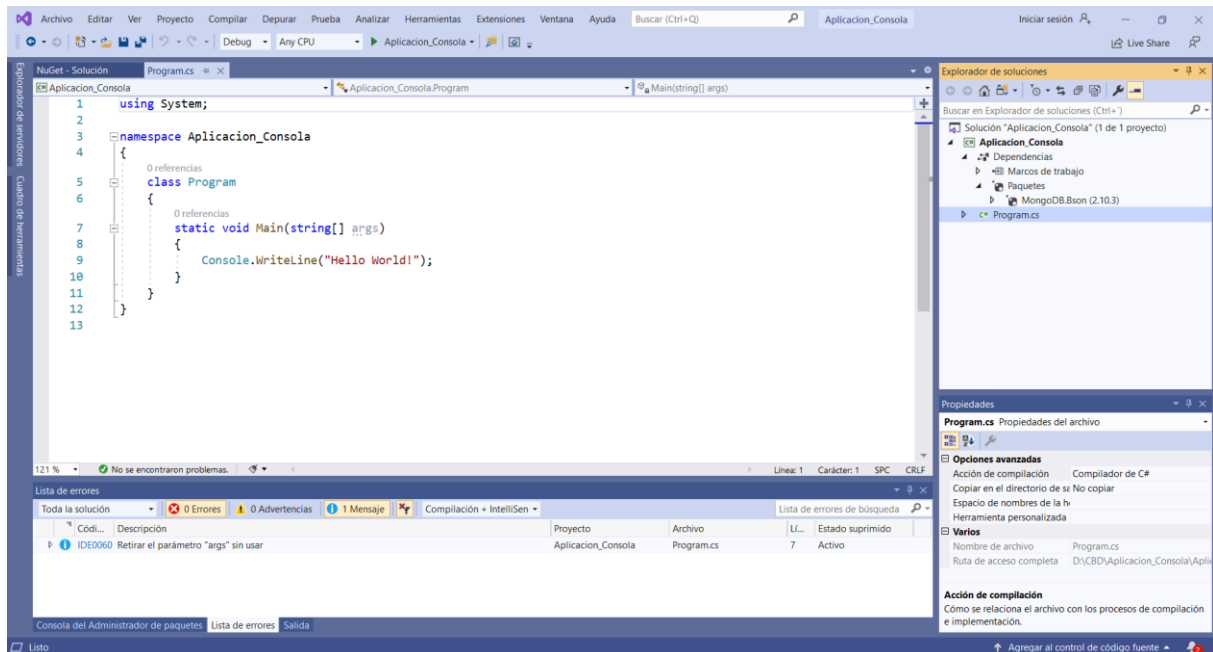


Ilustración 9. Proyecto Inicial con Drivers Instalados

Para comprobar que todo el proceso anterior se ha abordado con éxito comprobamos en las opciones de dependencias del proyecto que está instalado los paquetes Nuget.

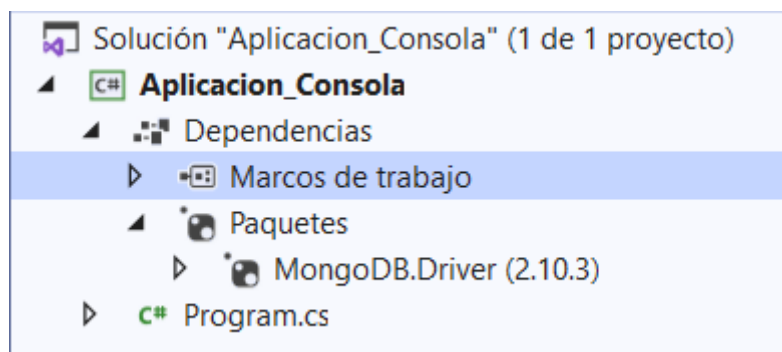


Ilustración 10. Comprobación Instalación Driver

9.2 Paso 2. Administrar conexión y operaciones con la Base de Datos

Una vez que tenemos creado nuestro proyecto en Visual Studio con las dependencias apropiadas para trabajar con MongoDB, vamos a conectar nuestra aplicación con la base de datos MongoDB.

9.2.1 Conectar la Base de Datos

Para ello vamos a crear una clase nueva, en nuestro caso hemos denominado `MongoConnect.cs`, en la que estableceremos la conexión con el servidor base de datos y se alojarán las operaciones con la misma, hemos decidido agrupar en esta clase, individual al programa principal, con todo lo referente con la base de datos para tener el código mejor organizado y accesible.

Con el fragmento que aparece a continuación establecemos la conexión con la base de datos en local, con el método `MongoConnect` que recibe el nombre la base de datos que quiere conectar, si esta no existe crea una nueva con ese nombre.

Código de la aplicación:

```
private IMongoDatabase db;

public MongoConnect(String database)
{
    var cliente = new MongoClient();
    db = cliente.GetDatabase(database);
}
```

El método `mongoConnect`, recibe un `String` en forma de la base de datos a la que queremos conectarlo, en el caso de que es nombre no exista creará una base de datos nueva, si ya existe se conectará a ella.

Dentro de ese método creamos un nuevo cliente de mongo con la sentencia

```
var cliente = new MongoClient();
```

Que nos permite establecer la conexión con nuestro servidor local de MongoDB, sabemos que se establece en local puesto que `new MongoClient()`, no recibe ningún parámetro por lo que por defecto establece conexión con nuestro servidor el local.

En el caso de que queramos establecer la conexión con otro servidor que no sea en local bastaría con especificar la dirección web de este servicio dentro de la sentencia `new MongoClient("<dirección del servidor de mongo desplegado>")`, siguiendo esta forma.

Una vez que hemos establecido conexión con la base de datos lo tenemos todo listo para comenzar a crear las operaciones CRUD, que nos permitan interactuar, con nuestra base de datos a través de la aplicación.

Destacamos una variable `private IMongoDatabase`, en el que almacenamos una instancia del servidor de la base de datos.

9.2.2 Insertar elementos de la Base de Datos

En este apartado vamos a definir como crear un método en C# que nos permita insertar elementos en nuestra base de datos de tipo MongoDB, a partir de un objeto previamente creado en C#, en concreto nuestro ejemplo va dirigido a Series. El método que aparece a continuación tiene un carácter genérico, es decir, a pesar de que en nuestro caso vaya dirigido hacia las series, no habría que realizar cambios en el caso que se reciba un tipo de objetos que no sea Series.

Código de la aplicación:

```
public void insertarElemento<T>(String coleccion, T elemento)
{
    var collection = db.GetCollection<T>(coleccion);
    collection.InsertOne(elemento);
}
```

Este método recibe dos parámetros el primer parámetro corresponde con la colección a la que queremos insertar ese objeto, si esa colección existe ya en nuestra base de datos, se insertaría un objeto en dicha colección, mientras que si no existiese en nuestra base de datos crearíamos una nueva colección con el nombre que introduzcamos. Como segundo parámetro el método recibe el elemento que queremos insertar, en este caso será un objeto de tipo Serie, pero se podría insertar cualquier otro tipo de objeto reutilizando todo este método gracias a ser genérico.

Una vez que tenemos establecido todos los parámetros bastaría con invocar el método InsertOne, de la librería MongoDB.driver el cual se encargará de insertar el elemento en nuestra base de datos.

9.2.3 Mostrar un elemento de la Base de Datos

A continuación, se muestra como mostrar un elemento de nuestra base de datos según sus parámetros, en este caso se muestra un objeto según su id de la base de datos.

Código de la aplicación:

```
public T mostrarElementoPorId<T>(String coleccion, ObjectId id)
{
    var collection = db.GetCollection<T>(coleccion);
    var filter = Builders<T>.Filter.Eq("_id", id);
    return collection.Find(filter).First();
}
```

Este método recibe dos parámetros la colección en la que queremos buscar ese objeto y su identificador del objeto de base de datos. Como primer paso primero buscamos la colección en la que queremos encontrar ese elemento para mostrarla, seguido a esto en la segunda línea creamos un filtro de modo que nos permita discriminar elementos según unos parámetros, en este caso hemos decidido eliminar a través de su Id, pero podríamos filtrar según los diferentes atributos del objeto de nuestra base de datos, es destacable, que el primer parámetro que recibe el filter es el nombre de atributo en base de datos en este caso "_id".

Una vez que tenemos establecido todos los parámetros bastaría con invocar el método Find, de la librería MongoDB.driver el cual se encargará de realizar toda la lógica de buscar el elemento y devolvérselo

9.2.4 Editar elementos de la Base de Datos

Siguiendo el esquema CRUD, a continuación, se muestra el código referente a modificar las características de un elemento que tenemos almacenado en base datos.

Código de la aplicación:

```
public void editarElemento<T>(String coleccion, T elemento, ObjectId id)
{
    var collection = db.GetCollection<T>(coleccion);
    var data = new BsonDocument("_id", id);
    collection.ReplaceOne(data, elemento, new ReplaceOptions { IsUpsert =
true });
}
```

Para modificar elementos, recibimos tres parámetros, la colección en la que queremos editar esos datos, el nuevo objeto y el identificador del objeto a editar.

Como primer paso idéntico a los métodos anteriores es explicitar en que colección de la base datos vamos a querer editar ese elemento, el siguiente paso es obtener ese elemento por id e introducirles las nuevas modificaciones, para ello usamos el método ReplaceOne de la librería MongoDB.driver, que nos permite editar elementos este método recibe tres parámetros, el primero recibe el objeto que se quiere modificar en formato Bson, seguido de este recibe el nuevo elemento con las modificaciones y finalmente unos parámetros que en de que si aparecen o no nos permite limitar el alcance del editado del elemento en cuestión, como nosotros no tenemos ningún inconveniente en su edición total lo hemos establecido a True.

Para obtener el objeto de la base de datos en formato Bson, hemos usado la segunda línea que nos permite obtener objetos en formato Bson, según alguno de sus atributos de la base de datos, al igual que en el filtro recibe el nombre de ese atributo en la base de datos en este caso “_id”

```
var data = new BsonDocument("_id", id);
```

9.2.5 Borrar elementos de la Base de Datos

Siguiendo el esquema de editar un elemento, primero obtenemos el elemento que queremos eliminar de la base de datos, y después le aplicamos la operación de eliminarlos en la base de datos, con ayuda del método de la librería MongoDB.Driver, DeleteOne.

Código de la aplicación:

```
public void eliminarElemento<T>(String coleccion, ObjectId id)
{
    var collection = db.GetCollection<T>(coleccion);
    var filter = Builders<T>.Filter.Eq("_id", id);
    collection.DeleteOne(filter);
}
```

9.2.6 Listar elementos de la Base de Datos

El siguiente método se encarga de mostrarnos todos los objetemos que tenemos almacenados en nuestra base de datos.

Código de la aplicación:

```
public List<T> listadoBD<T>(String coleccion)
{
    var collection = db.GetCollection<T>(coleccion);
    var data = new BsonDocument();
    return collection.Find(data).ToList();
}
```

9.2.7 Búsqueda de elementos en la Base de Datos

En el fragmento de código a continuación se muestra como poder buscar series según su título, a modo de un buscador que nos permita rápidamente a través del nombre mostrar esa serie.

Código de la aplicación:

```
public T filterByName<T>(String coleccion,String s)
{
    var collection = db.GetCollection<T>(coleccion);
    var filter = Builders<T>.Filter.Eq("titulo", s);
    return collection.Find(filter).FirstOrDefault();
}
```

9.3 Paso 3. Modelo de Datos a Insertar

Una vez que tenemos ya creada la conexión de nuestra base de datos y las operaciones a realizar en nuestra base de datos, vamos a crear el objeto que queremos insertar en nuestra base de datos, en este caso la aplicación va dirigida a manejar series de televisión por lo que procedemos a crear los modelos con los atributos referentes a series.

Este paso de crear objetos podría omitirse y en lugar de insertar objetos programáticos podríamos insertar objetos en formato Bson. Pero consideramos que es una buena práctica crear los objetos, ya que al usar C#, un lenguaje orientado a la programación de los orientada a objetos.

En el caso que nuestra aplicación vaya destinada a otro tipo de ejercicio que requiera un modelo diferente, podríamos crear el nuevo modelo siguiendo los pasos que se exponen a continuación, una vez elaborado el nuevo modelo, no habría que cambiar ningún método relacionado con las operaciones de inserción, edición, eliminación, puesto que son métodos que reciben cualquier tipo de dato y lo insertan en la base de datos.

Cómo la programación orientada a objetos el modelo se basa en definir los atributos que va a tener dicho objeto, seguido a esto las operaciones de get y de set para poder obtener y modificar los diferentes los diferentes atributos y finalmente los constructores que nos permiten crear ese objeto a partir de ciertos campos.

Finalmente, comenzamos a crear los modelos de Series y de Actores puesto que nuestra aplicación va destinada a Series y Actores

Código de la aplicación:

```
class Serie
{
    [BsonId]
    public ObjectId _id { get; set; }

    [BsonElement("Titulo")]
    public string titulo { get; set; }

    [BsonElement("Descripcion")]
    public string descripcion { get; set; }

    [BsonElement("Valoracion")]
    public double valoracion { get; set; }

    [BsonElement("Lanzamiento")]
    public String fechaLanzamiento { get; set; }

    public int temporada { get; set; }

    public Actor[] actores { get; set; }

    public Serie() {
    }
}
```

```

        public Serie(string tit, String des, double val, Actor[] actores, string
creador, String fechaLanzamiento, int temp )
        {
            this.titulo = tit;
            this.descripcion = des;
            this.valoracion = val;
            this.fechaLanzamiento = fechaLanzamiento;
            this.temporada = temp;
        }
    }
}

```

Como podemos apreciar aparecen las algunas notaciones:

- [BsonId], elemento de la librería MongoDB.Driver que nos permite que nos permite indicarle a Mongo que atributo va a ser serializado como el id, en la base datos, si no se establece MongoDB establece uno por defecto.
- [BsonElement("Titulo")], con esta cláusula podemos definir que nombre va a tener ese atributo en la base de datos, muy útil para redefinir el nombre de los diferentes atributos a la hora de almacenarlos en la base datos, puesto que puede encontrarse problemas al coincidir con palabras reservadas, elemento de la librería MongoDB.Driver

Atributos de modelo Serie y métodos get y set de dichos atributos

Código de la aplicación referente al Serie.cs :

```

[BsonId]
public ObjectId _id { get; set; }

[BsonElement("Titulo")]
public string titulo { get; set; }

[BsonElement("Descripcion")]
public string descripcion { get; set; }

[BsonElement("Valoracion")]
public double valoracion { get; set; }

```

Los constructores

Código de la aplicación referente al Serie.cs :

```
public Serie() {  
    }  
  
    public Serie(string tit, String des, double val, Actor[] actores, string  
creador, String fechaLanzamiento, int temp )  
    {  
        this.titulo = tit;  
        this.descripcion = des;  
        this.valoracion = val;  
        this.fechaLanzamiento = fechaLanzamiento;  
        this.temporada = temp;  
    }  
}
```

El modelo anterior también tiene asociado otro modelo que corresponde con los actores de las propias series en las que guardamos diferentes atributos.

Código de la aplicación referente al Actor.cs :

```
public class Actor  
{  
  
    public string nombre { get; set; }  
  
    public string apellidos { get; set; }  
  
    public string lugarNacimiento { get; set; }  
  
    public int edad { get; set; }  
  
    public double valoracion { get; set; }  
  
  
    public Actor()  
    {  
  
    }  
  
    public Actor(string nombre, string apellidos, string lugarNacimiento, int  
edad, double valoracion)  
    {  
  
    }  
}
```

```

        this.nombre = nombre;
        this.apellidos = apellidos;
        this.lugarNacimiento = lugarNacimiento;
        this.edad = edad;
        this.valoracion = valoracion;
    }

```

```

    }

```

Aquí de nuevo vemos otro nuevo ejemplo de programación orientada a objetos donde podemos ver los atributos de ese objeto y los métodos get y set:

Código de la aplicación referente al Actor.cs :

```

public string nombre { get; set; }

public string apellidos { get; set; }

public string lugarNacimiento { get; set; }

public int edad { get; set; }

public double valoracion { get; set; }

```

Los constructores:

Código de la aplicación referente al Actor.cs :

```

public Actor()
{
}

public Actor(string nombre, string apellidos, string lugarNacimiento, int
edad, double valoracion)
{
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.lugarNacimiento = lugarNacimiento;
    this.edad = edad;
    this.valoracion = valoracion;
}

```

9.4 Paso 4. Estilo

Una vez que hemos establecido ya nuestro modelo con el que queremos realizar operaciones en nuestra base de datos, así como las operaciones y las conexiones, lo tenemos todo listo para comenzar a elaborar el estilo de cómo vamos a presentar esa funcionalidad. En este caso hemos decidido enfocar la aplicación en una sencilla aplicación de consola que nos permita realizar simples operaciones con los objetos que hemos creado. Para ello nos apoyamos en las clases `Dialogo.cs`, `EstiloConsola.cs`, `Program.cs`.

Vamos a comenzar mostrando la clase `Dialogo.cs`, en la cual como su propio nombre indica establecemos las pautas que debe de tener la comunicación entre la consola y el usuario, para ello creamos métodos auxiliares que en función de la operación que pretendamos realizar nos muestre una serie de parámetros para poder realizar las operaciones que hemos definido previamente en Paso 2.

Para ello comenzamos estableciendo el estilo del menú de consola de bienvenida en el siguiente método:

Código de la aplicación:

```
public static int ShowMainMenu()
{
    int choice;

    Console.Clear();
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.SetCursorPosition(20, 0);
    Console.WriteLine("Bienvenido a nuestro trabajo de CBD; una app conectada
con MongoDB");
    Console.ResetColor();
    Console.WriteLine();
    Console.Write(
        "Por favor, introduzca su elección: \n\n" +
        "[0] Añadir nueva serie. \n" +
        "[1] Mostrar lista de series. \n" +
        "[2] Buscar serie. \n" +
        "[3] Mostrar las mejores series. \n" +
        "[4] Actualizar la información de una serie(por ID). \n" +
        "[5] Borrar serie (por ID). \n" +
        "[6] Salir. \n");
    Console.WriteLine("-----");

    var entry = Console.ReadLine();
    if (!int.TryParse(entry, out choice))
    {
        choice = 7;
    }
    return choice;
}
```

En las primeras líneas establecemos el título del trabajo, así como el color y finalmente mostramos al usuario las diferentes opciones que tiene en nuestra aplicación esta diferentes opciones tienen

asociadas una tecla numérica que nos permite en función de la tecla que pulsamos distinguir unas operaciones de otras.

También disponemos de otro menú que sigue la misma estructura en el que puedes insertar actores o no insertarlos en función de los aspectos que conozcas de ellas.

Código de la aplicación:

```
public static int ActorsMainMenu()
{
    int choice;

    Console.Clear();
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.SetCursorPosition(20, 0);
    Console.WriteLine("Menú de actores");
    Console.ResetColor();
    Console.WriteLine();
    Console.Write(
        "Por favor, introduzca su elección: \n\n" +
        "[0] Añadir nuevo actor. \n" +
        "[4] Salir. \n");
    Console.WriteLine("-----");

    var entry = Console.ReadLine();
    if (!int.TryParse(entry, out choice))
    {
        choice = 5;
    }
    return choice;
}
```

Una vez que hemos generado ya la vista en la que aparecen las operaciones que nuestra aplicación puede realizar, a través de los menús, faltaría unir esos menús con las diferentes operaciones que realiza nuestra aplicación en la base de datos. Vamos a pasar a mostrar los diferentes métodos de la clase Dialogo.cs que se encargan de pedirnos la información necesaria para que cada una de las diferentes funcionalidades implementadas en la aplicación reciba la información necesaria para que se pueda ejecutar correctamente.

Código de la aplicación:

```
public static Serie ShowAddNewSerie()
{
    ShowHeader("Añadir nueva serie");

    var serie = new Serie();

    Console.Write("Introduzca el título de la serie: ");
    serie.titulo = Console.ReadLine();

    Console.Write("Introduzca una descripción: ");
    serie.descripcion = Console.ReadLine();

    Console.Write("Introduzca su valoración: ");
}
```

```

        double d;
        while (!double.TryParse(Console.ReadLine(), out d))
        {
            Console.WriteLine("El valor ingresado no es válido.\nIngrese un número
entero: ");
        }

        serie.valoracion = d;

        Console.WriteLine("Introduzca fecha de lanzamiento, siguiendo el formato
dd/MM/yyyy: ");
        serie.fechaLanzamiento = Console.ReadLine();
        DateTime dt;
        while (!DateTime.TryParseExact(serie.fechaLanzamiento, "dd/MM/yyyy", null,
System.Globalization.DateTimeStyles.None, out dt))
        {
            Console.WriteLine("Invalid date, please retry, the format is
dd/MM/yyyy");
            serie.fechaLanzamiento = Console.ReadLine();
        }

        Console.WriteLine("Introduzca el número de temporadas: ");
        // int num;

        // while (!Int32.TryParse(serie.temperada, null, null,
System.Globalization.DateTimeStyles.None, out num)){

        int num;

        while (!Int32.TryParse(Console.ReadLine(), out num))
        {
            Console.WriteLine("El valor ingresado no es válido.\nIngrese un número
entero: ");
        }

        serie.temperada = num;

        Console.WriteLine("Introduzca los actores de la serie: ");
        serie.actores=añadirActor();

        return serie;
    }

    public static Actor[] añadirActor()
    {
        ShowHeader("Añadir actores a la serie");
        List<Actor> actores = new List<Actor>();
        int menuChoice;
        do{
            menuChoice = ActorsMainMenu();

            switch (menuChoice) {

```

```
        case 0:
            var actor = new Actor();

            Console.WriteLine("Introduzca el nombre del actor: ");
            actor.nombre = Console.ReadLine();

            Console.WriteLine("Introduzca los apellidos: ");
            actor.apellidos = Console.ReadLine();

            Console.WriteLine("Introduzca su valoración: ");
            double num;

            while (!double.TryParse(Console.ReadLine(), out num))
            {
                Console.WriteLine("El valor ingresado no es válido.\nIngrese
un número decimal: ");
            }

            // actor.valoracion = Double.Parse(Console.ReadLine());

            Console.WriteLine("Introduzca la edad: ");
            int numero;

            while (!Int32.TryParse(Console.ReadLine(), out numero))
            {
                Console.WriteLine("El valor ingresado no es válido.\nIngrese
un número entero: ");
            }

            actor.edad = numero;

            Console.WriteLine("Introduzca el lugar de nacimiento: ");
            actor.lugarNacimiento = Console.ReadLine();

            actores.Add(actor);

            ShowActorAddedMessage();
            break;
    }

    } while (menuChoice != 4);

    return actores.ToArray();
}
```

Una vez mostrado el código de cómo va a ser el diálogo del usuario y la máquina para insertar actores, con sus correspondientes actores vamos a pasar a comentar su estructura, para ello destacamos mensajes auxiliares para ayudarnos al usuario a manejarse por la aplicación a través de mensajes que sepan en todo momento en qué lugar de la aplicación se encuentran. Como por ejemplo `ShowHeader("Añadir nueva serie");`, que nos indica que estamos añadiendo una serie en este momento.

Una vez nos disponemos a introducir los diferentes valores de los atributos de actor, los parámetros se introducen de manera secuencial, es decir, la aplicación nos guía en todo momento de qué campo estamos introduciendo y qué valor queremos introducirle a ese atributo, para facilitarle la labor al usuario, cada campo tiene sus respectivas validaciones para que en caso de error de algún campo pueda volverse introducirse.

Quiero destacar que el método añadir serie, devuelve un objeto de tipo serie que después será el que decidamos introducir en nuestra base de datos

Estas validaciones las realizamos a través del siguiente fragmento de código, en este caso es una validación de que la valoración sea de tipo numérica

Código de la aplicación:

```
Console.WriteLine("Introduzca su valoración: ");

double d;
while (!double.TryParse(Console.ReadLine(), out d))
{
    Console.WriteLine("El valor ingresado no es válido.\nIngrese un número entero: ");
}

serie.valoracion = d;
```

Aquí vemos las validaciones de fechas

Código de la aplicación:

```
Console.WriteLine("Introduzca fecha de lanzamiento, siguiendo el formato dd/MM/yyyy: ");
serie.fechaLanzamiento = Console.ReadLine();
DateTime dt;
while (!DateTime.TryParseExact(serie.fechaLanzamiento, "dd/MM/yyyy", null, System.Globalization.DateTimeStyles.None, out dt))
{
    Console.WriteLine("Invalid date, please retry, the format is dd/MM/yyyy");
    serie.fechaLanzamiento = Console.ReadLine();
}
```

Las validaciones de números enteros

Código de la aplicación:

```

Console.Write("Introduzca el número de temporadas: ");
// int num;

// while (!Int32.TryParse(serie.temporada,null,null,
System.Globalization.DateTimeStyles.None, out num)){

    int num;

    while (!Int32.TryParse(Console.ReadLine(), out num))
    {
        Console.Write("El valor ingresado no es válido.\nIngrese un número
entero: ");
    }

    serie.temporada = num;

```

Finalmente, el método de insertar series llama a un método auxiliar de insertar actores, el cual se encarga de llevarnos a un menú el cual nos permite insertar los diferentes actores que pertenecen a esa serie. La inserción de actores se realiza a través de un método auxiliar puesto que el modelo de series espera una lista de Actores, por lo que el método encargado de insertar actores se encarga de guardar esos actores que introducimos secuencialmente, agruparlos en una lista e insértalos en la base de datos.

Código de la aplicación:

```

public static Actor[] añadirActor()
{
    ShowHeader("Añadir actores a la serie");
    List<Actor> actores = new List<Actor>();
    int menuChoice;
    do{
        menuChoice = ActorsMainMenu();

        switch (menuChoice) {

            case 0:
                var actor = new Actor();

                Console.Write("Introduzca el nombre del actor: ");
                actor.nombre = Console.ReadLine();

                Console.Write("Introduzca los apellidos: ");
                actor.apellidos = Console.ReadLine();

                Console.Write("Introduzca su valoración: ");
                double num;

```

```

        while (!double.TryParse(Console.ReadLine(), out num))
        {
            Console.WriteLine("El valor ingresado no es válido.\nIngrese
un número decimal: ");
        }

        // actor.valoracion = Double.Parse(Console.ReadLine());

        Console.WriteLine("Introduzca la edad: ");
        int numero;

        while (!Int32.TryParse(Console.ReadLine(), out numero))
        {
            Console.WriteLine("El valor ingresado no es válido.\nIngrese
un número entero: ");
        }

        actor.edad = numero;

        Console.WriteLine("Introduzca el lugar de nacimiento: ");
        actor.lugarNacimiento = Console.ReadLine();

        actores.Add(actor);

        ShowActorAddedMessage();
        break;
    }

    } while (menuChoice != 4);

    return actores.ToArray();
}

```

Siguiendo con la estructura creamos diferentes métodos en función de cada una de las operaciones de base de datos que disponemos, ahora le tocaría el turno al método que nos permite listar las series que tenemos almacenadas en nuestra base de datos.

Código de la aplicación:

```

public static void ShowSerieList(List<Serie> seriesList)
{
    ShowHeader("Lista de series");

    var table = new EstiloConsola("Id", "Titulo", "Descripcion",
"Valoracion", "Temporadas", "Lanzamiento");

    foreach (var serie in seriesList)
    {
        table.AddRow(serie._id, serie.titulo, serie.descripcion,
serie.valoracion, serie.temporada, serie.fechaLanzamiento);
    }
    table.Print();
}

```

```

        ShowContinueMessage();
    }

```

Aquí vemos el fragmento que se encarga de listarlos diferentes objetos, para presentar los objetos de una forma que se adapte al formato de consola y que permita diferenciar los diferentes campos de los objetos que tenemos introducidos en la base de datos, para ello nos apoyamos en la clase auxiliar EstiloConsola.cs.

Código de la aplicación de EstiloConsola.cs:

```

class EstiloConsola
{
    private readonly string[] titles;
    private readonly List<int> lengths;
    private readonly List<string[]> rows = new List<string[]>();

    public EstiloConsola(params string[] titles)
    {
        this.titles = titles;
        lengths = titles.Select(t => t.Length).ToList();

        /// <summary>
        /// Add row to table with consideration of its content length
        /// </summary>
        /// <param name="row"></param>
        public void AddRow(params object[] row)
        {
            if (row.Length != titles.Length)
            {
                throw new System.Exception($"Added row length [{row.Length}] is not
equal to title row length [{titles.Length}]");
            }
            rows.Add(row.Select(o => o.ToString()).ToArray());
            for (int i = 0; i < titles.Length; i++)
            {
                if (rows.Last()[i].Length > lengths[i])
                {
                    lengths[i] = rows.Last()[i].Length;
                }
            }
        }

        /// <summary>
        /// Display pretty table on the screen
        /// </summary>
        public void Print()
        {
            lengths.ForEach(l => System.Console.Write("+-" + new string('-', l) + '-
'));
            System.Console.WriteLine("+");

            string line = "";
            for (int i = 0; i < titles.Length; i++)
            {

```


Continuamos explorando la clase Dialogo.cs con las operaciones de eliminar y editar

Código de la aplicación:

```
public static string ShowUpdateSerie()
{
    ShowHeader("Actualizar serie");

    Console.WriteLine("Introduzca serie Id: ");

    return Console.ReadLine();
}

/// <summary>
/// Display 'Delete serie' dialog
/// </summary>
/// <returns></returns>
public static string ShowDeleteSerie()
{
    ShowHeader("Borrar serie");

    Console.Write("Introduzca serie ID: ");

    return Console.ReadLine();
}
```

El fragmento de código se encarga de pedirnos el id del objeto que queremos eliminar para proceder a su eliminación o edición.

El resto de código de Dialogo.cs, al igual que listar los diferentes objetos de la base de datos se encarga de mostrarnos en formato de tabla el resultado de diferentes búsquedas que realizamos en nuestra base de datos.

Código de la aplicación:

```
public static void ShowSerieList(List<Serie> seriesList)
{
    ShowHeader("Lista de series");

    var table = new EstiloConsola("Id", "Titulo", "Descripcion",
    "Valoracion", "Temporadas", "Lanzamiento");

    foreach (var serie in seriesList)
    {
        table.AddRow(serie._id, serie.titulo, serie.descripcion,
        serie.valoracion, serie.temporada, serie.fechaLanzamiento);
    }
}
```

```

        }
        table.Print();

        ShowContinueMessage();
    }

    public static void ShowSerieListMejores(List<Serie> seriesList)
    {
        ShowHeader("Lista de las mejores series");

        var table = new EstiloConsola("Id", "Titulo", "Descripcion", "Valoracion",
"Temporadas", "Lanzamiento");

        foreach (var serie in seriesList)
        {
            table.AddRow(serie._id, serie.titulo, serie.descripcion,
serie.valoracion, serie.temporada, serie.fechaLanzamiento);
        }
        table.Print();

        ShowContinueMessage();
    }
    public static void ShowSerieByName(Serie serie)
    {
        ShowHeader("SERIE");

        var table = new EstiloConsola("Id", "Titulo", "Descripcion", "Valoracion",
"Temporadas", "Lanzamiento");

        table.AddRow(serie._id, serie.titulo, serie.descripcion, serie.valoracion,
serie.temporada, serie.fechaLanzamiento);

        table.Print();

        ShowContinueMessage();
    }
}

```

Finalmente, una vez que ya tenemos establecido el dialogo, tenemos todo listo para comenzar a montar nuestro programa ejecutable para ello, nos situamos ahora en la clase Program.cs, dentro del método main.

Código de la aplicación:

```

static void Main(string[] args)
{
    // Establecemos conexion con la base de datos
    var mongoDB = new MongoConnect("Entrega12");

    const string collectionName = "Series";
}

```

```
Console.Title = "Aplicación de consola de Mongo con C#, esperamos que os
guste...";

int menuChoice;

do
{
    menuChoice = Dialogo.ShowMainMenu();
    switch (menuChoice)
    {
        case 0: // Add new serie
        {
            var elemento = Dialogo.ShowAddNewSerie();

            mongoDB.insertarElemento(collectionName, elemento);

            Dialogo.ShowContinueMessage();
        }
        break;
        case 1: // Show serie list
        {
            var seriesList = mongoDB.listadoBD<Serie>(collectionName);
            Dialogo.ShowSerieList(seriesList);

        }
        break;
        case 2: // Show filter name
        {

            Dialogo.ShowSerieByName1();

            String ser= Console.ReadLine();

            Serie s= mongoDB.filterByName<Serie>(collectionName, ser);
            while (s == null)
            {
                Console.WriteLine("No tenemos esta serie, introduzca otra:");

                String ser2 = Console.ReadLine();
                Serie s2 = mongoDB.filterByName<Serie>(collectionName,
ser2);

                if(s2!=null)
                    Dialogo.ShowSerieByName(s2);
            }

            Dialogo.ShowSerieByName(s);

        }
        break;

        case 3: // Show serie list mejores
        {
            var seriesList =
mongoDB.listadoBestBD<Serie>(collectionName);
```

```

        Dialogo.ShowSerieListMejores(seriesList);
    }
    break;

case 4: // Update serie info (by ID)
{
    var serieId = Dialogo.ShowUpdateSerie();
    ObjectId serieIdGuid;

    bool isValidGuid = ObjectId.TryParse(serieId, out
serieIdGuid);

    if (isValidGuid)
    {
        var serie =
mongoDB.mostrarElementoPorId<Serie>(collectionName, serieIdGuid);
        Console.WriteLine($"{\n" +
            $"Current info\n" +
            $"Título: {serie.titulo} \n" +
            $"Descripción: {serie.descripcion}");

        Console.WriteLine("\n-----\n");

        Console.WriteLine("Introduzca un nuevo título (leave
empty if no changes");
        var titulo = Console.ReadLine();

        if (!string.IsNullOrEmpty(titulo) &&
!string.IsNullOrWhiteSpace(titulo))
        {
            serie.titulo = titulo;
        }

        Console.WriteLine("Introduzca una nueva descripcion
(leave empty if no changes");
        var description = Console.ReadLine();

        if (!string.IsNullOrEmpty(description) &&
!string.IsNullOrWhiteSpace(description))
        {
            serie.descripcion = description;
        }

        Console.WriteLine("Introduzca una nueva valoracion
(leave empty if no changes");
        var valoracion = Console.ReadLine();

        if (!string.IsNullOrEmpty(valoracion) &&
!string.IsNullOrWhiteSpace(valoracion))
        {
            serie.valoracion = Double.Parse(valoracion);
        }
    }
}

```

```

        Console.WriteLine("Introduzca una nueva fecha de
Lanzamiento, según el patrón dd-mm-yyyy: (leave empty if no changes)");
        serie.fechaLanzamiento = Console.ReadLine();
        DateTime dt;
        while (!DateTime.TryParseExact(serie.fechaLanzamiento,
"dd/MM/yyyy", null, System.Globalization.DateTimeStyles.None, out dt))
        {
            Console.WriteLine("Invalid date, please retry");
            serie.fechaLanzamiento = Console.ReadLine();
        }

        Console.WriteLine("Introduzca el número de temporadas:
(leave empty if no changes)");
        int numero;

        while (!Int32.TryParse(Console.ReadLine(), out
numero))
        {
            Console.Write("El valor ingresado no es
válido.\nIngrese un número entero: ");
        }

        Console.WriteLine("Introduzca los actores de la serie:
(leave empty if no changes), si quiere introducir presine una tecla y enter");
        var actor = Console.ReadLine();

        if (!string.IsNullOrEmpty(actor) &&
!string.IsNullOrWhiteSpace(actor))
        {
            serie.actores = Dialogo.añadirActor();
        }

        mongoDB.editarElemento<Serie>(collectionName, serie,
serieIdGuid);
    }
    else
    {
        Console.WriteLine($"Exception: '{serieId}' is not a
valid Guid!");
    }

    Dialogo.ShowContinueMessage();
}
break;
case 5: // Delete serie (by ID)
{
    var serieId = Dialogo.ShowDeleteSerie();

    ObjectId serieIdGuid;

    bool isValidGuid = ObjectId.TryParse(serieId, out
serieIdGuid);

    if (isValidGuid)
    {

```

```

                                mongoDB.eliminarElemento<Serie>(collectionName,
serieIdGuid);
                                }
                                else
                                {
                                    Console.WriteLine($"Exception: '{serieId}' is not a
valid Guid!");
                                }

                                Dialogo.ShowContinueMessage();
                            }
                            break;
                        }
                    } while (menuChoice != 6);
                }
            }
        }
    }
}

```

Primero destacamos una variable `mongoDB`, que hace referencia al nombre de la base de datos que hemos creado, en el caso de que no tengamos ninguna, se creará una base de datos con ese nombre y en el caso de que exista nos conectaremos a la base de datos que tenga dicho nombre, si queremos cambiar la base de datos y conectarnos a otra bastaría simplemente con cambiar ese parámetro por el nombre de nuestra nueva base de datos. También he de destacar que tenemos que especificar el nombre de la colección a la que le queremos realizar las operaciones, destacar que al igual que la base de datos si esa colección no existe se crea una nueva mientras que si existe nos conectaremos a ella, si queremos cambiar la colección a la que queremos realizarle las operaciones bastaría con cambiar la línea `collectionName`.

A modo de resumen en estas dos variables establecemos a que base de datos nos vamos a conectar o crear y que colección vamos a crear o centrarnos

Código de la aplicación:

```

// Establecemos el nombre de la colección a la que le queremos aplicar
las diferente operaciones
const string collectionName = "Series";

// Establecemos conexión con la base de datos
var mongoDB = new MongoConnect("Entrega12");

```

Seguido de esto, establecemos un switch para que en función de parámetro que elijamos se realice una operación u otra, así como unimos la funcionalidad creada en la clase `MongoConnect.cs` junto con el dialogo que hemos establecido en la clase `Dialogo.cs` de la siguiente forma:

- Comenzamos llamando a `Dialogo.cs`, para comenzar la comunicación entre el usuario y la aplicación

- Una vez se han comunicado y se ha decidido realizar una operación y otra, procedemos a realizar la operación a través de la clase MongoConnect.cs

Este patrón se repite a lo largo de las diferentes funcionalidades de la aplicación.

Código de la aplicación:

```
case 0: // Add new serie
    {
        var elemento = Dialogo.ShowAddNewSerie();

        mongoDB.insertarElemento(collectionName, elemento);

        Dialogo.ShowContinueMessage();
    }
    break;
```

Como citamos anteriormente el método ShowAddedNewSerie de la clase Dialogo.cs devuelve un objeto de tipo Serie, que para que persista en nuestra base de datos tenemos que insertarlo a través del método insertar que previamente creamos en la clase MongoConnect.cs.

En el caso de la edición a parte de las acciones ya mencionadas anteriormente también se lleva a cabo una validación de los campos. Así como una pequeña comprobación de que el id del objeto que vamos a eliminar existe en nuestra base de datos.

Código de la aplicación :

```
case 4: // Update serie info (by ID)
    {
        var serieId = Dialogo.ShowUpdateSerie();
        ObjectId serieIdGuid;

        bool isValidGuid = ObjectId.TryParse(serieId, out
serieIdGuid);

        if (isValidGuid)
        {
            var serie =
mongoDB.mostrarElementoPorId<Serie>(collectionName, serieIdGuid);
            Console.WriteLine($"{\n" +
                $"Current info\n" +
                $"Título: {serie.titulo} \n" +
                $"Descripción: {serie.descripcion}");

            Console.WriteLine("\n-----
-----\n");

            Console.WriteLine("Introduzca un nuevo título (leave
empty if no changes");
```

```

        var titulo = Console.ReadLine();

        if (!string.IsNullOrEmpty(titulo) &&
!string.IsNullOrWhiteSpace(titulo))
        {
            serie.titulo = titulo;
        }

        Console.WriteLine("Introduzca una nueva descripcion
(leave empty if no changes)");
        var description = Console.ReadLine();

        if (!string.IsNullOrEmpty(description) &&
!string.IsNullOrWhiteSpace(description))
        {
            serie.descripcion = description;
        }

        Console.WriteLine("Introduzca una nueva valoracion
(leave empty if no changes)");
        var valoracion = Console.ReadLine();

        if (!string.IsNullOrEmpty(valoracion) &&
!string.IsNullOrWhiteSpace(valoracion))
        {
            serie.valoracion = Double.Parse(valoracion);
        }

        Console.WriteLine("Introduzca una nueva fecha de
Lanzamiento, según el patrón dd-mm-yyyy: (leave empty if no changes)");
        serie.fechaLanzamiento = Console.ReadLine();
        DateTime dt;
        while (!DateTime.TryParseExact(serie.fechaLanzamiento,
"dd/MM/yyyy", null, System.Globalization.DateTimeStyles.None, out dt))
        {
            Console.WriteLine("Invalid date, please retry");
            serie.fechaLanzamiento = Console.ReadLine();
        }

        Console.WriteLine("Introduzca el número de temporadas:
(leave empty if no changes)");
        int numero;

        while (!Int32.TryParse(Console.ReadLine(), out
numero))
        {
            Console.Write("El valor ingresado no es
válido.\nIngrese un número entero: ");
        }

        Console.WriteLine("Introduzca los actores de la serie:
(leave empty if no changes), si quiere introducir presine una tecla y enter");
        var actor = Console.ReadLine();

        if (!string.IsNullOrEmpty(actor) &&
!string.IsNullOrWhiteSpace(actor))
        {

```



```
        serie.actores = Dialogo.añadirActor();  
    }  
  
    mongoDB.editarElemento<Serie>(collectionName, serie,  
serieIdGuid);  
    }  
    else  
    {  
        Console.WriteLine($"Exception: '{serieId}' is not a  
valid Guid!");  
    }  
  
    Dialogo.ShowContinueMessage();  
}
```

Para el resto de los métodos de la clase Program.cs, los métodos siguen con el patrón anterior, hacemos llamada a la clase dialogo con el correspondiente método y finalmente para reflejar esos cambios usamos la clase MongoConnect.cs

10 Problemas

Los problemas que hemos encontrado principalmente era el desconocimiento de la tecnología, el desarrollo de la aplicación ha sido realizado sin conocimientos previos de la tecnología C#, lo que nos ha ralentizado el proceso de creación de la aplicación, así como varias dudas en el desarrollo sobre como implementar parte de la aplicación, especialmente encontramos problemas con las validaciones de objetos en la base de datos, para abordar este problema, buscamos a lo largo de la documentación de C# y MongoDB y destinándonos ambos miembros de la pareja de desarrollo a esa tarea en cuestión de realizar las validaciones, finalmente conseguimos abordar el problema y llegamos a una solución, que se corresponde a las líneas de código relacionadas con la validación de los diferentes campos de series y actores.

También hemos obtenido problemas a la hora de poder desplegar una base de datos de tipo Mongo Atlas, no hemos podido establecer la conexión con dicha base de datos, pretendíamos como idea tener una base de datos disponible en la nube pero su integración con el proyecto ha sido dificultoso y requería más tiempo del previsto, así como un esfuerzo mayor y la documentación que encontrábamos sobre el problema no nos sirvió de ayuda, ante esta situación decidimos evadir esta tarea, y utilizar el servidor local de MongoDB, en lugar de otra solución en la nube.

Finalmente, encontramos dificultades, a la hora de reunirnos y planificar, ya que éramos una pareja que nunca habíamos trabajado conjuntamente anteriormente, pero pese a las dificultades, a través de vías telemáticas, conseguimos rápidamente ponernos al día y estar en contacto.

11 Mejoras

En este apartado enunciamos posibles mejoras que podríamos realizarle a la aplicación en un futuro para que esta adquiera un mayor valor, entre ellas destacamos:

- Disponibilidad de la aplicación en la web
- Inserción de datos, a través de listas
- Inserción de tipos genéricos
- Evolucionar la aplicación y crear una interfaz gráfica, en lugar de una interfaz de comandos

12 Conclusiones

Tras la realización de la aplicación, hemos aprendido nociones básicas de como integrar dos tecnologías muy potentes como son MongoDB y C#, así mismo, hemos adquirido conocimientos del lenguaje C#. A todo esto, se suma una introducción a aplicaciones basadas en bases de datos NoSQL, al haber usado MongoDB.

También destacamos la facilidad de integración de ambas tecnologías, así como su poder de escalabilidad, así como la adaptación al teletrabajo debido a la situación actual.

Podemos destacar que la realización del proyecto ha sido positiva ya que hemos adquirido nuevos conocimientos, gracias a introducirnos a nuevas tecnologías como MongoDB y C#.

13 Anexo

13.1 Links de interés

- <https://docs.mongodb.com/manual/crud/>, manual de Mongo CRUD
- <https://www.mongodb.com/cloud/atlas>, URL Mongo Atlas
- <https://docs.mongodb.com/manual/tutorial/getting-started/>, Introducción a MongoDB
- <https://docs.microsoft.com/es-es/dotnet/csharp/getting-started/>, Introducción a C#
- https://www.w3schools.com/cs/cs_getstarted.asp, Introducción a C#
- <https://www.youtube.com/playlist?list=PLIIMS2xXqAXmTC2nuxajMjShNXtqQILn1>, Lista de reproducción de la App

13.2 Instalar Visual Studio

En este apartado describiremos como descargar e instalar Visual Studio

1. Accedemos a la web de Microsoft para proceder a la descarga de Visual Studio:

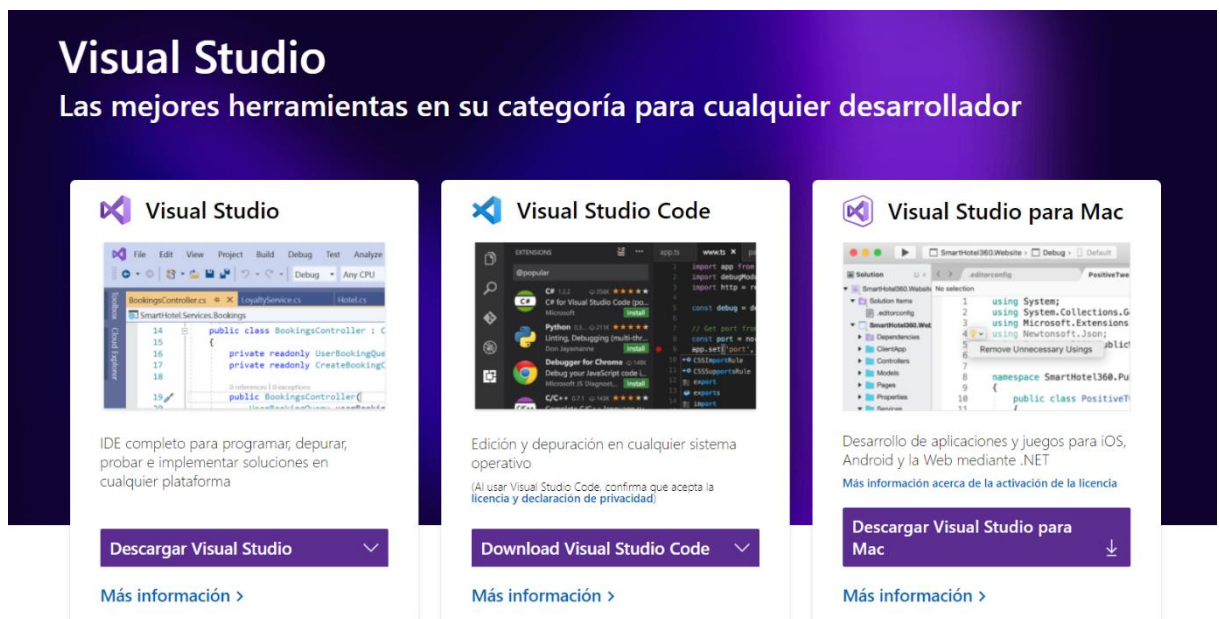


Ilustración 12. Visual Studio instalación

En nuestro caso elegimos la primera opción, pero todo depende del sistema operativo que tengamos.

Completamos la descarga e instalamos, procedemos a asignarle un espacio en el disco duro y ya está listo para usarse

<https://visualstudio.microsoft.com/es/?rr=https%3A%2F%2Fmedium.com%2F%40hstechup%2Fcreating-your-first-c-application-with-mongodb-2f2fbbfa661f>

13.3 Instalar MongoDB

1. Ahora procedemos a la instalación de MongoDB:

Entrar en la web de mongoDB ([mongodb.org](https://www.mongodb.com)) y descargar el archivo de instalación para Windows 64 bit: MongoDB 4.2.3, <https://www.mongodb.com/download-center/community>.

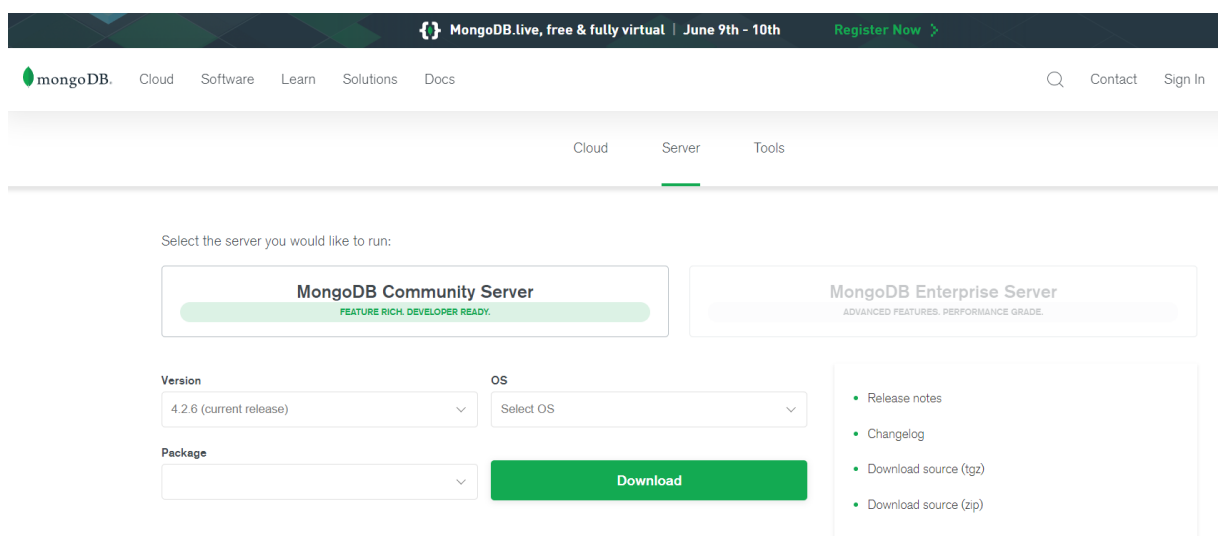


Ilustración 13. MongoDB instalación

1.2. Crear un directorio en c:\data\db, donde se alojarán los datos (es posible elegir otra). Eso es el dbpath en mongoDB.

1.3. Ejecutar el archivo de instalación en modo “Complete”

1.4. MongoDB se encuentra funcionando en el puerto 27017, por defecto. Y la consola de administración en el 28017.

1.5. En el directorio de c:\data\db estarán: El fichero mongod.lock que almacena el PID del proceso mongod. Los ficheros WiredTiger. El directorio Journal para recuperación de datos en caso de pérdida. El directorio diagnostic.data es un log de estadísticas del servidor.

3. Para arrancar el servidor ejecutamos una línea de comandos en MS-DOS con cmd luego nos dirigimos al directorio donde se ha instalado (por defecto, C:\Archivos de programa\mongodb\Server\3.6\bin). Ejecutar mongod.

4. Arrancar el cliente (shell), ejecutamos en otra ventana una línea de comandos MSDOS luego nos dirigimos al directorio donde se ha instalado (C:\Archivos de programa\mongodb\Server\3.6\bin). Por último ejecutamos mongo y utilizamos el comando Show dbs que muestra las bases de datos existentes, aparecen las bases de datos “admin”, “config” y “local”.

13.4 Código de la aplicación

Para la instalación del proyecto clonar el repositorio de Github con la url, <https://github.com/juanluisperez1/CBD>, la aplicación se encuentra en la rama master

El proyecto consta de una carpeta .vs donde se guardan ficheros de configuración de la herramienta Visual Studio, un directorio denominado CBD-PROYECTO, en el que se encuentran los diferentes archivos que componen la aplicación y un fichero con extensión .sln, que corresponde con la extensión que Visual Studio les da a sus proyectos.

Para abrirlo deberíamos, abrir un proyecto existente con Visual Studio y abrir el fichero con extensión .sln, siguiendo las directrices de Apartado 8 de documento.