Clase 2 - DA2 tecnologias

@March 11, 2023

En esta clase mostraremos la creación de una web API con ASP .NET Core.

Para esto se va a necesitar las siguientes herramientas y tecnologias:

- ASP.NET 6
- **Visual Studio Code o Visual Studio**
- **Postman**

Extensiones recomendadas para mejorar el trabajo con VSCode

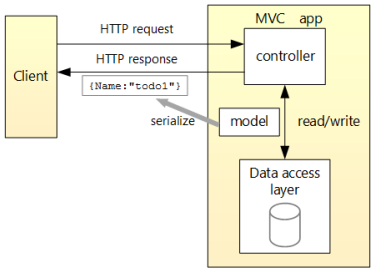
- .NET Core Extension Pack
- Rainbow Brackets

Esta clase vamos a ver como:

- Crear un proyecto WebApi
- Crear un controller con métodos CRUD
- Configurar endpoints y valores de retorno
- Llamar la WebApi con Postman
- Navegar por las demas capas, business logic, data acces y domain

Para realizar todo lo mencionado anteriormente, vamos a crear una API llamada Cinema la cual se encargara de almacenar películas y actores.

Diseño de la app



Creación del proyecto

- Abrir **Visual Studio Code**
- Abrir una terminal y dirigirse hacia la carpeta donde quieren crear el proyecto

Comandos a tener en cuenta

Lista de comandos a tener en cuenta para el proyecto

Siempre que se necesite ayuda con el comando o no se sabe cual usar, la mejor ayuda es correr `dotnet [COMANDO] -h` para una explicacion de los parametros del comando o cual es su funcionalidad. Tambien se puede correr `dotnet -h` para poder ver los comandos disponibles

Commando	Resultado
`dotnet new sln`	Creamos solucion (principalmente útil para VisualStudio, cuando queremos abrir la solución y levantar los proyectos asociados)
`dotnet new webapi -n "Nombre del Proyecto"`	Crear un nuevo Proyecto del template WebApi
`dotnet sln add`	Asociamos el proyecto creado al .sln
`dotnet sln list`	Vemos todos los proyectos asociados a la solución
`dotnet new classlib -n "Nombre del Proyecto"`	Crear un nueva libreria (standard)
`dotnet add "Nombre del Proyecto 1".csproj reference "Nombre del Proyecto 2".csproj`	Agrega una referencia al Proyecto 1 del Proyecto 2
`dotnet add package "Nombre del Package"`	Instala el Package al proyecto actual. Similar a cuando se agregaban paquetes de Nuget en .NET Framework.
`dotnet build`	Compilar y generar los archivos prontos para ser deplegados (*production build*)
`dotnet run`	Compilar y correr el proyecto

1. Creamos la solución

```
dotnet new sln -n Cinema
```

2. Creamos un proyecto de tipo WebAPI

```
dotnet new webapi -n "Cinema. WebApi"
```

3. Agregamos el proyecto a la solución

```
dotnet sln add Cinema.WebApi
```

MOSTRAR COMO QUEDA LA ESTRUCUTRA

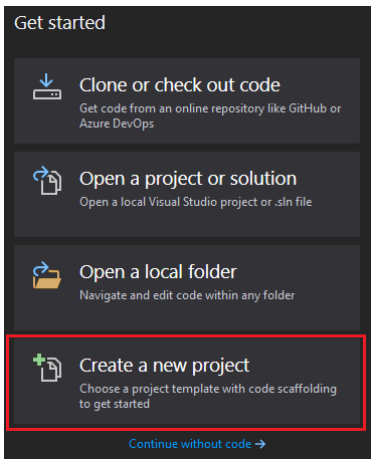
Es necesario crear una solucion? (SLN)

No, no es necesario. Sin embargo, tener una solucion permite crear/compilar/manejar todos los proyectos involucrados juntos, sin tener que correr cada uno. Se maneja todo como una unica unidad.

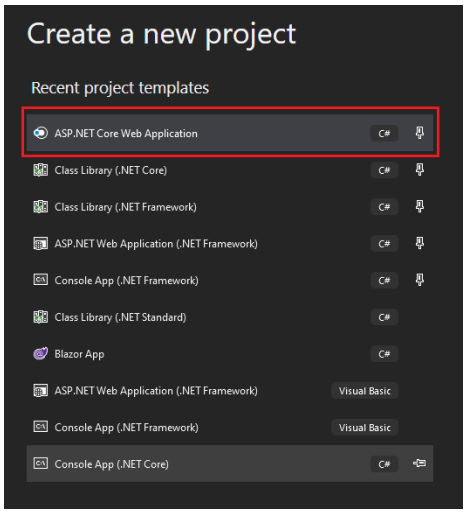
LES MUESTRO RIDER

En el caso de usar Visual Studio

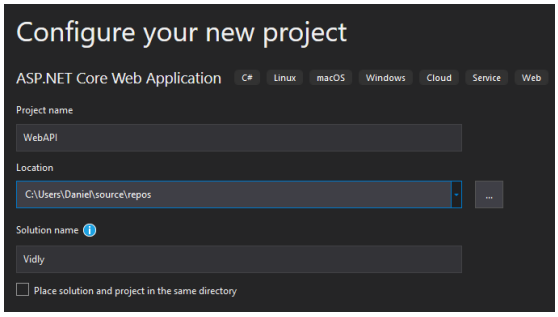
- Abrir Visual Studio
- Seleccionar **Crear un nuevo proyecto**



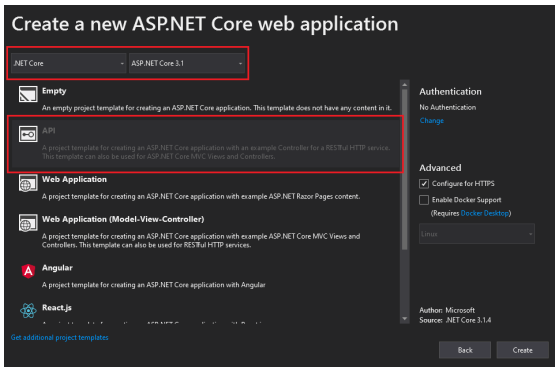
- Seleccionar **ASP.NET Core Web Application** en los diferentes templates que se les muestra y apretar **Next**



- Escribir el nombre del proyecto **WebApi** y en la solución **Cinema** y seleccionen **Create**



- En el dialogo **Create a new ASP.NET Core web Application,** confirmen que **.NET Core** y **ASP.NET 6** esta seleccionado.
- Seleccionen el template de **API** y seleccionen **Create**



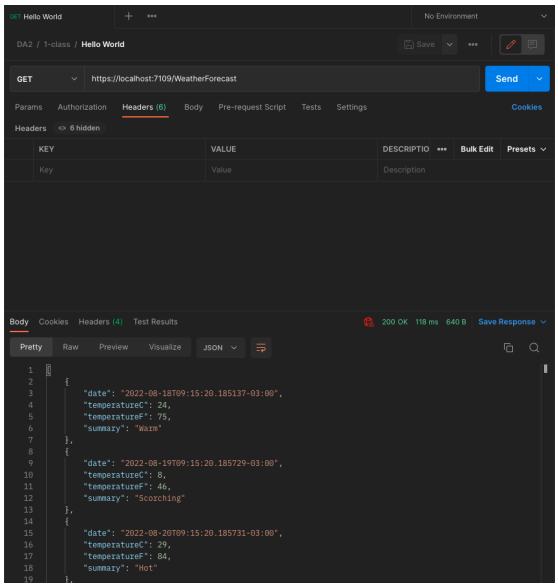
4. En ambos escenarios vamos a ver lo mismo, vamos a encontrar un template con un controlador. Para probar la **api** en **Visual Studio Code** se tiene que correr el comando

```
cd cinema.WebApi
dotnet run
```

eso abrirá un puerto por defecto al cual le deberíamos de mandar las request y para **Visual Studio** basta con apretar **F5** o el **botón de play.**

5. Vemos en postman el endpoint creado por defecto

```
https://localhost:7109/WeatherForecast
```



Vamos a recorrer la solución y a analizar ciertos puntos de la misma

Program.cs

Es el punto de entrada de nuestra aplicación donde configuramos el Web host. Esta clase configura aspectos de la infraestructura de la aplicación tales como Web host, Logging, Contenedor de Inyecciones de Dependencias,etc. La aplicación es creada, configurada y construida usando el método `CreateBuilder` en la clase Program.

Esta clase es similar a la clase Program de una aplicación de consola, y eso es porque todas las aplicaciones .NET Core son básicamente aplicaciones de consola.

El principal objetivo de esta clase es configurar la infraestructura de la aplicación.

Al inicio, esta clase crea un Web Host, una vez creado, se le configura un montón de elementos.

Web Host

El web host es el responsable de inicializar la aplicación y correrla. Los web hosts crean un servidor, el cual escucha request HTTP en un determinado puerto libre de la maquina. Es el encargado de configurar la request pipeline. También es el encargado de setear el contenedor de servicios donde nosotros podemos agregar nuestros propios servicios a ser usados, eso también incluye controlar el ciclo de vida de los servicios según el tipo de ciclo de vida seteado por nosotros.

En resumen, un Web Host, deja lista nuestra aplicación para recibir request, pero, este Web Host tiene que ser creado y configurado en la clase Program.

El WebServer usado por default en una api .NET Core es Kestrel. Kestrel es un HTTP Server open-source y cross-platform. Fue desarrollada para hostear aplicaciones .NET Core en cualquier plataforma.

Nuestro primer controlador

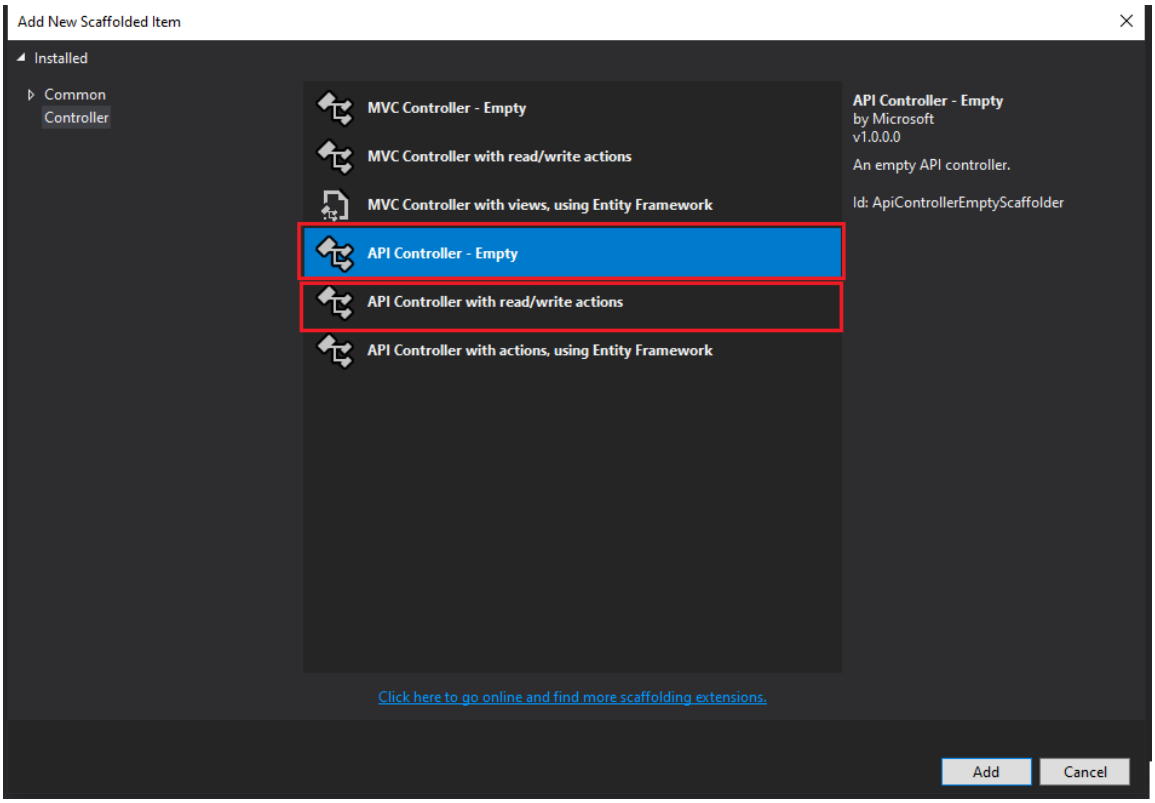
Un controlador es una clase que hereda de **ControllerBase**. Una **WebApi** consiste en tener uno o varias de estas clases.

ControllerBase provee muchas properties y métodos que son útiles para manejar requests HTTP.

Por ejemplo **CreatedAction** retorna el código de error **201**, **BadRequest** retorna **400**, **NotFound** retorna **404**, entre otros. Para mas información sobre los diferentes métodos y properties vean **ControllerBase Class**

Entonces, para crear nuestro primer controlador hacemos click derecho en la carpeta **controllers** que podemos encontrar en el proyecto **WebApi** y hacemos lo siguiente:

- Para **Visual Studio Code:**
  - Seleccionamos la opción **New File**
  - Escribimos el nombre **MovieController.cs.** La extension es necesaria.
- Para **Visual Studio:**
  - Seleccionamos la opción **Add** → **Controller** y nos mostrara la siguiente ventana



  - Podemos seleccionar la opción **API Controller - Empty** la cual nos va a crear una clase controller vacía o la opción **API Controller with read/write actions** la cual nos va a crear una clase controller con las operaciones **CRUD.**
  - Una vez seleccionada la opcion se le pone el nombre **MovieController.cs,** aca no es necesario especificar la extension.

Listo, ya tenemos nuestro primer controlador para exponer el recurso **Movie** al mundo.

Antes de escribir los **verbos HTTP** y unos cuantos métodos, veamos que son los **attributes**.

Attributes

Los attributes pueden ser usados para configurar el comportamiento de los controladores de la web API.

el [ApiController] attribute

se especifica en la clase controlador para permitir los siguientes comportamientos:

- Obligatorio el uso del attribute [Route]
- Respuestas HTTP 400 automaticas
- El uso de binding de parámetros
- Detalle de problemas para códigos de error

Una forma de evitar utilizar **[ApiController]** en todos nuestros controladores es realizar un controlador base al cual se le adjudique este atributo y todos nuestros controladores heredarían de este.

```
[ApiController]
public class CinemaControllerBase : ControllerBase
{
}
```

```
[Route("api/movies")]
public class MovieController : CinemaControllerBase
{
}
```

Otra opción que podemos optar es aplicar este attribute a nivel de assembly sin necesidad de aplicarlo a nivel de clase lo cual permitirá que todos los controladores se les aplique. Seria de la siguiente manera en la clase **Startup**:

```
[assembly: ApiController]
namespace WebApi
{
  public class Startup
  {
    ...
  }
}
```

Attributes para endpoints

Los siguientes attributes serán usados para especificar que verbos HTTP nuestro controlador soporta:

Verbos HTTP

Aa Verbo	Attribute	Property
GET	[HttpGet] o [HttpGet(string template)]	Identifica una acción que soporta el verbo GET de HTTP
POST	[HttpPost] o [HttpPost(string template)]	Identifica una acción que soporta el verbo POST de HTTP
PUT	[HttpPut] o [HttpPut(string template)]	Identifica una acción que soporta el verbo PUT de HTTP
DELETE	[HttpDelete] o [HttpDelete(string template)]	Identifica una acción que soporta el verbo DELETE de HTTP

Como podemos observar en la tabla de arriba cada verbo tiene asignado dos attributes, uno que no recibe nada y otro que recibe por parámetro un string. Ese string que le pasamos servirá para pasar valores por la uri (**[HttpGet("{id}")]** → ***api/movies/5***) o para crear un nivel mas sobre la ruta (**[HttpGet("premiere")]** → ***api/movies/premiere***), claramente hay una diferencia entre estos dos parámetros.

Attributes en los parámetros

Estos attributes especifican la ubicación en donde se encuentra el valor del parámetro en la request. Los que vamos a usar son los siguientes:

- **[FromBody]** → Viene un body en la request
- **[FromHeader]** → Vienen headers en la request
- **[FromQuery]** → Vienen parámetros en la request
- **[FromRoute]** → Viene en la misma request la información

Queremos crear endpoints para las siguientes acciones para lo que antes de meternos a crearlos en el controllador vamos a pensar como construirlos teniendo en cuenta lo visto anteriormente

1. Traer todas las películas que se encuentren en el sistema: GET *api/movies*

2. Traer todas las películas que sean mayores 18 que se encuentren en el sistema:

   Vamos a usar metodo GET *api/movies?ageAllowed=18 y por query la edad*

3. Traer la película con identificador 5:

   Vamos a usar metodo GET *api/movies/5 y por route el id*

4. Crear una película: POST *api/movies*
5. Actualizar la película con identificador 5: PUT *api/movies/5*
6. Eliminar la película con identificador 5: DELETE *api/movies/5*
7. Eliminar todas las películas en el sistema: DELETE *api/movies*

Ahora vamos a crear estos endpoint, para esto solo se mostrará el attribute, la firma del método y se retornara 200 Ok:

1. Traer todas las películas que se encuentren en el sistema: GET *api/movies*

```
[HttpGet]
0 references
public IActionResult Get()
{
    return Ok();
}
```

2. Traer todas las películas que sean mayores 18 que se encuentren en el sistema: GET *api/movies?ageAllowed=18*

```
//api/movies?ageAllowed=5
[HttpGet]
0 references
public IActionResult GetBy([FromQuery]int ageAllowed)
{
    return Ok();
}
```

3. Traer la película con identificador 5: GET *api/movies/5*

```
//api/movies/5
[HttpGet("{id}")]
0 references
public void Get([FromRoute]int id)
{

}
```

4. Crear una película: POST *api/movies*

```
//api/movies
[HttpPost]
0 references
public IActionResult Post([FromBody]MovieModel movie)
{
    return Ok();
}
```

5. Actualizar la película con identificador 5: PUT *api/movies/5*

```
//api/movies/5
[HttpPut("{id}")]
0 references
public IActionResult Put([FromRoute]int id, [FromBody]MovieModel movie)
{
    return Ok();
}
```

6. Eliminar la película con identificador 5: DELETE *api/movies/5*

```
//api/movies/5
[HttpDelete("{id}")]
0 references
public IActionResult Delete([FromRoute]int id)
{
    return Ok();
}
```

7. Eliminar todas las películas en el sistema: DELETE *api/movies*

```
//api/movies
[HttpDelete]
0 references
public IActionResult Delete()
{
    return Ok();
}
```

IActionResult es adecuado para situaciones en las que se necesita la flexibilidad de devolver diferentes tipos de resultados.

```
using Microsoft.AspNetCore.Mvc;
```

```
namespace Cinema.WebApi;

[ApiController]
[Route("api/movies")]
public class MovieController:ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return Ok();
    }

   [HttpGet("ageAllowed")]
    public IActionResult GetByAge([FromQuery] int ageAllowed)
    {
        return Ok();
    }

    [HttpGet("{id}")]
    public IActionResult GetById([FromRoute] int id)
    {
        return Ok();
    }

    [HttpPost]
    public IActionResult Post([FromBody] MovieModel movie)
    {

        return Ok();
    }

    [HttpPut("{id}")]
    public IActionResult Put([FromRoute] int id,[FromBody] MovieModel movie)
    {

        return Ok();
    }

    [HttpDelete("{id}")]
    public IActionResult Delete([FromRoute] int id)
    {
        return Ok();
    }

    [HttpDelete]
    public IActionResult Delete()
    {
        return Ok();
    }
}
```

```
namespace Cinema.WebApi;

public class MovieModel
{
```

```
}
```

Un detalle es que se puede mappear los valores de la request con los tipos en los parametros gracias al mecanismo **Model Binding** que nos provee **ASP.NET Core.**

Model Binding es la automatización del proceso de convertir el string que escriben los clientes a los tipos de **.NET** correspondientes.

Lo que hace es:

- Saca la información de varios lugares, del body, header, route, query strings.
- Esta información se la da a los controladores en forma de parámetros en métodos o properties publicas.

Para saber mas sobre **Model Binding** lean: **Model Binding in ASP.NET Core**

Que tanto exponer?

Veamos una buena practica de las APIs y relacionado también a la seguridad. Si se fijan el tipo de parámetro tanto para el método ***POST*** y ***PUT*** es **MovieModel,** el cual es un representante de la entidad **Movie,** donde esta es la entidad que se persiste en la base de datos y la que es manejada por la lógica de negocio.

Si en vez de recibir **MovieModel** recibiéramos **Movie,** estaríamos exponiendo toda la información de la misma. Las APIs en producción por lo general limitan la data que reciben y devuelven utilizando un modelo que tiene un sub-set de la entidad. Nos vamos a referir a esta técnica como **Data Transfer Object (DTO)**, **input model**, o **view model**.

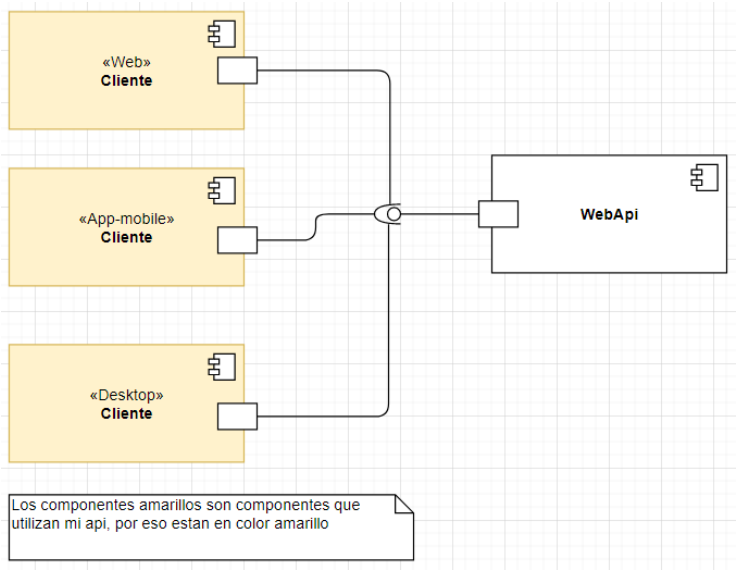
Estos modelos son usados para:

- Prevenir mostrar de mas
- Esconder properties a clientes que no se suponen que vean
- Omitir algunas properties para reducir tamaño
- Minimizar las relaciones entre objetos.
- Minimizar el impacto de cambio a los clientes

Cual es el objetivo de los modelos?

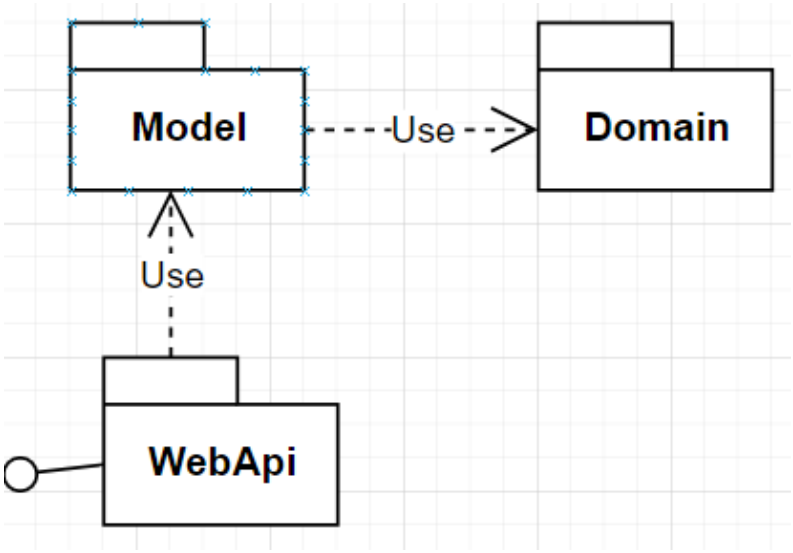
La realización de **modelos** o **dto** es la implementacion del **patron Adapter.** De esta forma estamos **envolviendo** las entidades para que los diferentes clientes no se vean afectados si las entidades cambian su estado. Veamos algunos diagramas:

Diagrama de componentes y conectores



En esta diagrama podemos observar como diferentes clientes se acoplan a mi **WebApi** y esta es la que escucha sus peticiones. Este tipo de vista es dinámica porque muestra componentes que corren en RAM, que consumen tiempo en el procesador.

Diagrama de paquetes



En este diagrama vemos como el paquete **WebApi** que es donde se encuentran todos los controladores que son el punto de entrada a los diferentes clientes, solamente se acopla al paquete **Model** y solamente entiende estas clases y no las persistidas que se encuentran en **Domain.**

El impacto de cambio de las clases de **Domain** va en sentido inverso a la flecha de los paquetes que dependen de el, esto significa que solamente impactaría en **Model,** solo se propagaría hasta **Model** y **WebApi** no se vería afectado. Como esos cambios no se

propagan mas lejos de **Model** los clientes que utilizan nuestra **api** no se ven afectados de tener que actualizar sus modelos que reciben por la **api**.

En conclusion, la **api** solamente conoce modelos, mientras que la **lógica** de negocio solo conoce **entidades**. De esta forma ayudamos a los clientes a que usen lo necesario y no utilicen modelos estables.

Una vez tenemos creado el proyecto WebAPI vamos a ver como crear los demas proyectos para nuestra solucion

Creamos la libreria **businesslogic** y la agregamos al sln

```
dotnet new classlib -n Cinema.BusinessLogic
dotnet sln add Cinema.BusinessLogic
```

Creamos la libreria **dataaccess** y la agregamos al sln

```
dotnet new classlib -n Cinema.DataAccess
dotnet sln add Cinema.DataAccess
```

Creamos la libreria **domain** y la agregamos al sln

```
dotnet new classlib -n Cinema.Domain
dotnet sln add Cinema.Domain
```

Vamos a agregar la referencia desde webAPI a DataAccess, Domain y BusinessLogic

```
dotnet add Cinema.WebApi reference Cinema.DataAccess
dotnet add Cinema.WebApi reference Cinema.Domain
dotnet add Cinema.WebApi reference Cinema.BusinessLogic
```

Agregamos la referencia desde DataAccess a Domain

```
dotnet add Cinema.DataAccess reference Cinema.Domain
```

Agregamos la referencia desde businessLogic a Domain y DataAccess

```
dotnet add Cinema.BusinessLogic reference Cinema.Domain
dotnet add Cinema.BusinessLogic reference Cinema.DataAccess
```

Vamos a crear dentro de **Movie.Domain** la clase que representa a nuestras peliculas **Movie**

```
namespace Cinema.domain;

public class Movie
{
        public Guid Id { get; set; }
        public string Name { get; set; }
        public DateTime Date { get; set; }
        public string Description { get; set; }
        public int AgeAllowed { get; set; }
        public List<Actor> Actors { get; set; }

        public Movie()
        {
            Id = Guid.NewGuid();
            Actors = new List<Actor>();
        }
}
```

Nuestros "Peliculas" **Movie** tienen una lista de Actores. Debido a esto, tambien debemos crear la clase **Actor**

```
namespace Cinema.domain;

public class Actor
{
    public class Movie
    {
        public Guid Id { get; set; }
        public string Name { get; set; }
        public DateTime Date { get; set; }
        public string FilmGenre { get; set; }
        public List<Actor> Actors { get; set; }
```

```
        public Movie()
        {
            Id = Guid.NewGuid();
            Actors = new List<Actor>();
        }
    }
}
```

Como vemos el `Id`, es una instancia de Guid. El método Guid.NewGuid() crea un identificador único universal (UUID) de la versión 4 como se describe en RFC 4122, sec. 4.4 . Se garantiza que el Guid devuelto no es igual a Guid.Empty .

Dentro de Cinema.DataAccess

Dentro del proyecto de DataAccess, crearemos MoviesRepository. Este seria el encargado de devolvernos los datos necesarios que tengamos guardados. Como no tenemos logica alguna, simplemente crearemos un par de objetos *dummy* y los devolveremos en una lista.

El codigo es bastante directo, simplemente creamos dos Actors y dos Movies

```
using Cinema.domain;

namespace cinema.DataAccess;
public class MoviesRepository
{
    public List<Movie> GetMovies() {
        List<Movie> movies = new List<Movie>();

        Actor firstActor = CreateActor("firstActor", "firstFilmGenre");
        Actor secondActor = CreateActor("secondActor", "firstFilmGenre");

        Movie firstMovie = CreateMovie("firstName","firstDescription", 18,firstActor);
        movies.Add(firstMovie);

        Movie secondMovie = CreateMovie("secondName","secondDescription",10, secondActor);
        movies.Add(secondMovie);

        return movies;
    }

    private Actor CreateActor(String name, string filmGenre) {
        Actor actor = new Actor();
        actor.Name = name;
        actor.FilmGenre = filmGenre;
        return actor;
    }
```

```
    private Movie CreateMovie(String name, String description, int ageAllowed, Actor actor) {
        Movie movie = new Movie();
        movie.Actors.Add(actor);
        movie.Name = name;
        movie.Description = description;
        movie.AgeAllowed = ageAllowed;
        return movie;
    }
}
```

Dentro de Cinema.BusinessLogic

Crearemos una clase llamada **MoviesLogic**, la cual tiene la logica de

Logic:

```
    public class MoviesLogic
    {
        private MoviesRepository moviesRepository;

        public MoviesLogic() {
            moviesRepository = new MoviesRepository();
        }

        public List<Movie> GetMovies() {
            return moviesRepository.GetMovies();
        }
    }
```

Conectando todo

Ahora es hora de conectar todo. Haremos que **MoviesController** tenga una instancia de **MoviesLogic**, y que esta ultima tenga una instancia de MoviesRepository. Cada una llamara al metodo de cada clase.

MovieController*:*

```
[Route("api/movies")]
[ApiController]
public class MoviesController : ControllerBase
{
    private MoviesLogic moviesLogic;

    public MoviesController() {
        moviesLogic = new MoviesLogic();
    }
```

```
    // GET api/movies
    [HttpGet]
    public ActionResult Get()
    {
        List<Movie> movies = moviesLogic.GetMovies();
        return Ok(movies);
    }
}
```

Ahora vamos a postman y ejecutamos el endpoint y

```
https://localhost:7144/api/movies
```

Vemos que nos devuelve lo siguiente