



Guia creacion WebApi

En esta guía se mostrara la creación de una web API con ASP .NET Core.

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

- **ASP.NET Core 3.1:** <https://dotnet.microsoft.com/download/dotnet-core/3.1>
- **Visual Studio Code** (<https://code.visualstudio.com/download>) o **Visula Studio** (https://portal.azure.com/?Microsoft_Azure_Education_correlationId=175771eb-1548-444e-9394-db6546f1cb2e#blade/Microsoft_Azure_Education/EducationMenuBlade/software)
- **Postman:** <https://www.postman.com/downloads/>

Para aquellas personas que utilicen **Visual Studio Code** va a ser necesario que instalen las siguientes extensions:

- .NET Core Extension Pack (obligatorio)
- Bracket Pair Colorizer (opcional)





Previamente dicho esto empecemos con la guia.

Algunos de los puntos que se mostrara son los siguientes:

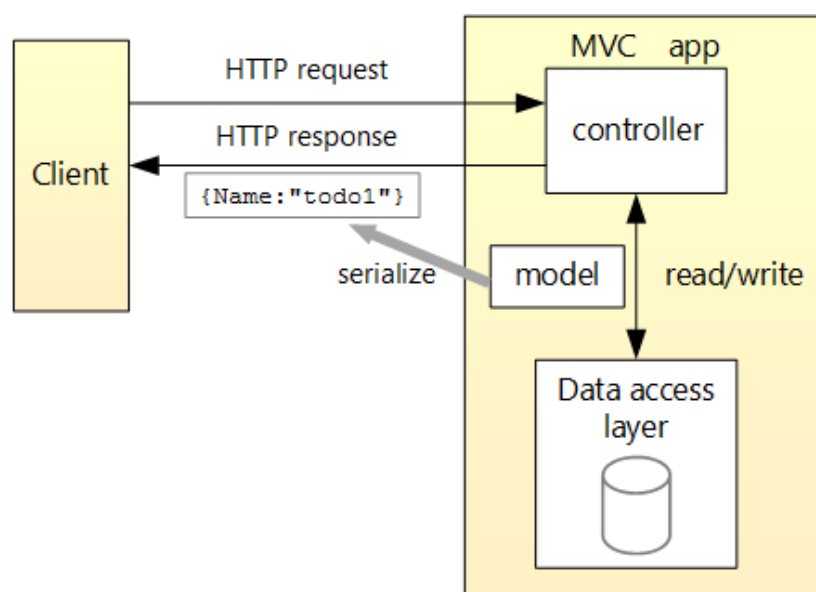
- Crear un proyecto WebApi
- Crear un controller con métodos CRUD
- Configurar endpoints y valores de retorno
- Llamar la WebApi con Postman

Para realizar todo lo mencionado anteriormente, vamos a crear una API llamada Vidly la cual se encargara de almacenar películas. Para ello vamos a diseñar primeramente los siguientes endpoints:

Endpoints

 API	 Description	 Request body	 Response body
<u>GET</u> <u>/api/movies</u>	Traer todas las peliculas	-	Una lista de todas las peliculas
<u>GET</u> <u>/api/movies/{id}</u>	Traer una pelicula por el ID	-	La pelicula con el {id}
<u>GET</u> <u>/api/movies?</u> <u>{ageAllowed}</u>	Traer todas las peliculas que sean mayores a la edad permitida	-	Una lista de todas las peliculas que cumplen con la condicion
<u>POST</u> <u>/api/movies</u>	Crear una pelicula	Pelicula	Pelicula con identificador
<u>PUT</u> <u>/api/movies/{id}</u>	Actualizar una pelicula existente	Pelicula	-
<u>DELETE</u> <u>/api/movies/{id}</u>	Eliminar una pelicula	-	-

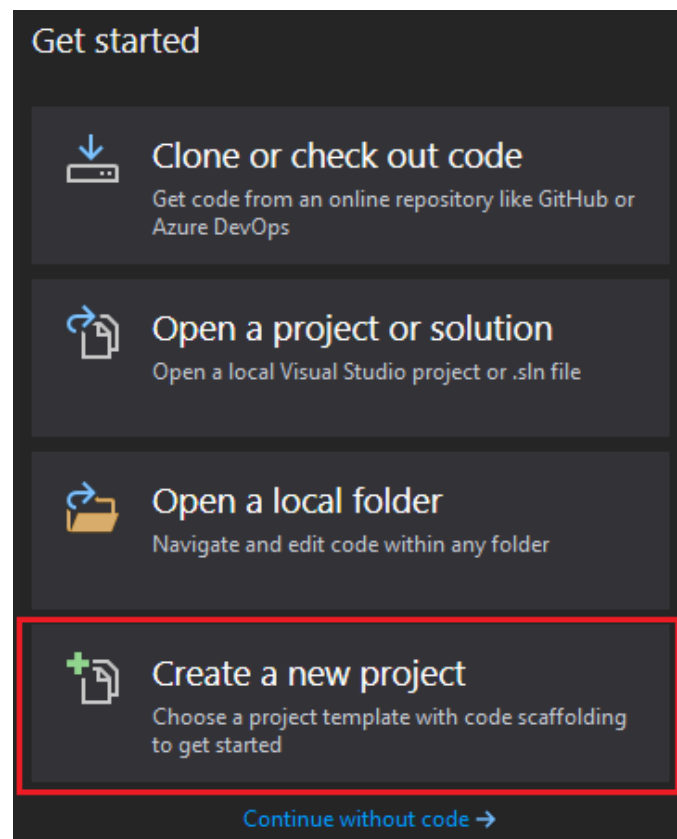
A continuación se mostrara un diagrama que muestra el diseño de la app:



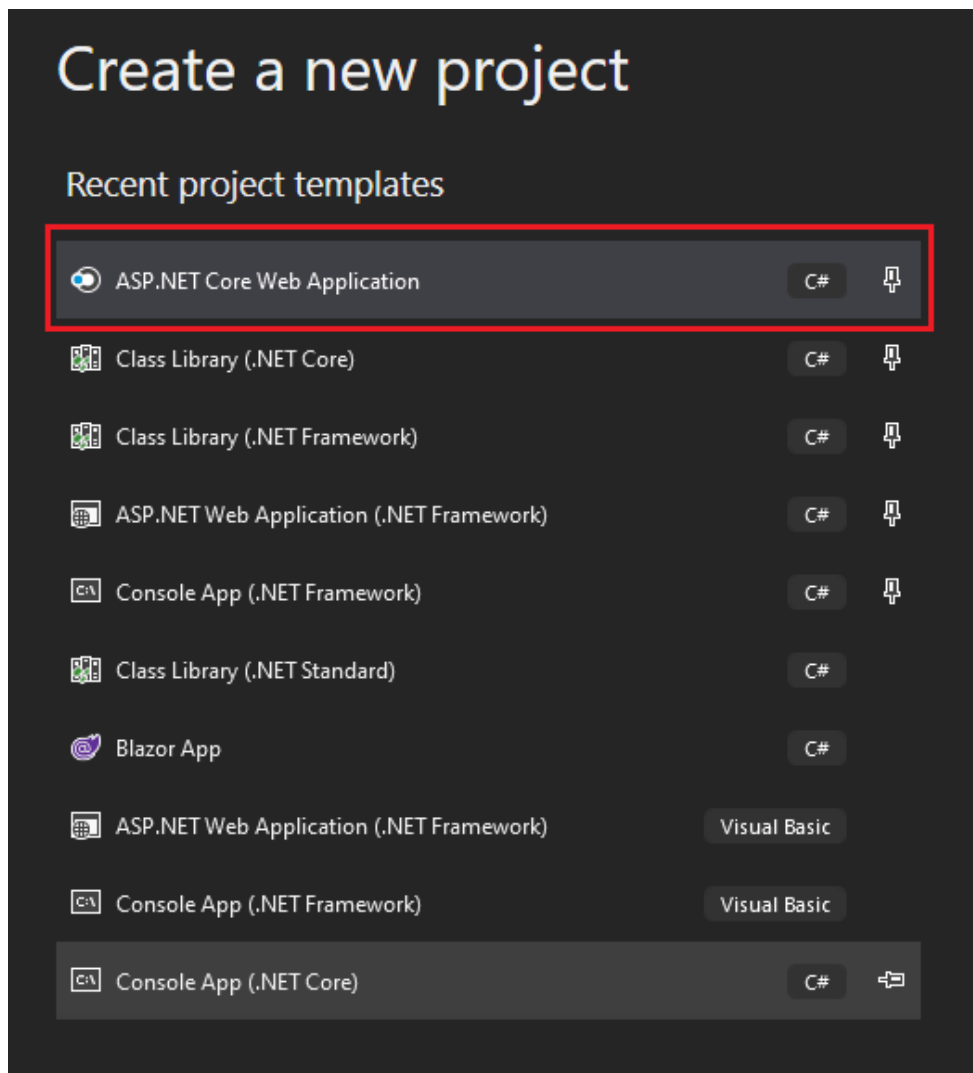
Creación del proyecto

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 **Vidly** y seleccionen **Create**

Configure your new project

ASP.NET Core Web Application C# Linux macOS Windows Cloud Service Web

Project name

WebAPI

Location

C:\Users\Daniel\source\repos

Solution name ⓘ

Vidly

☐ Place solution and project in the same directory

- En el dialogo **Create a new ASP.NET Core web Application**, confirmen que **.NET Core** y **ASP.NET Core 3.1** esta seleccionado.
- Seleccionen el template de **API** y seleccionen **Create**

Create a new ASP.NET Core web application

.NET Core ASP.NET Core 3.1

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Angular
A project template for creating an ASP.NET Core application with Angular

React.js
A project template for creating an ASP.NET Core application with React.js

[Get additional project templates](#)

Authentication
No Authentication
[Change](#)

Advanced
☒ Configure for HTTPS
☐ Enable Docker Support
(Requires [Docker Desktop](#))
Linux

Author: Microsoft
Source: .NET Core 3.1.4

Back Create

Visual Studio Code

- Abrir **Visual Studio Code**
- Abrir una terminal y dirigirse hacia la carpeta donde quieren crear el proyecto
- Una vez ahí, escribir los siguientes comandos:

```
dotnet new sln -n Vidly
dotnet new webapi -au none -n WebApi
dotnet sln add WebApi
```

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 **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**.

Nuestro primer controlador

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

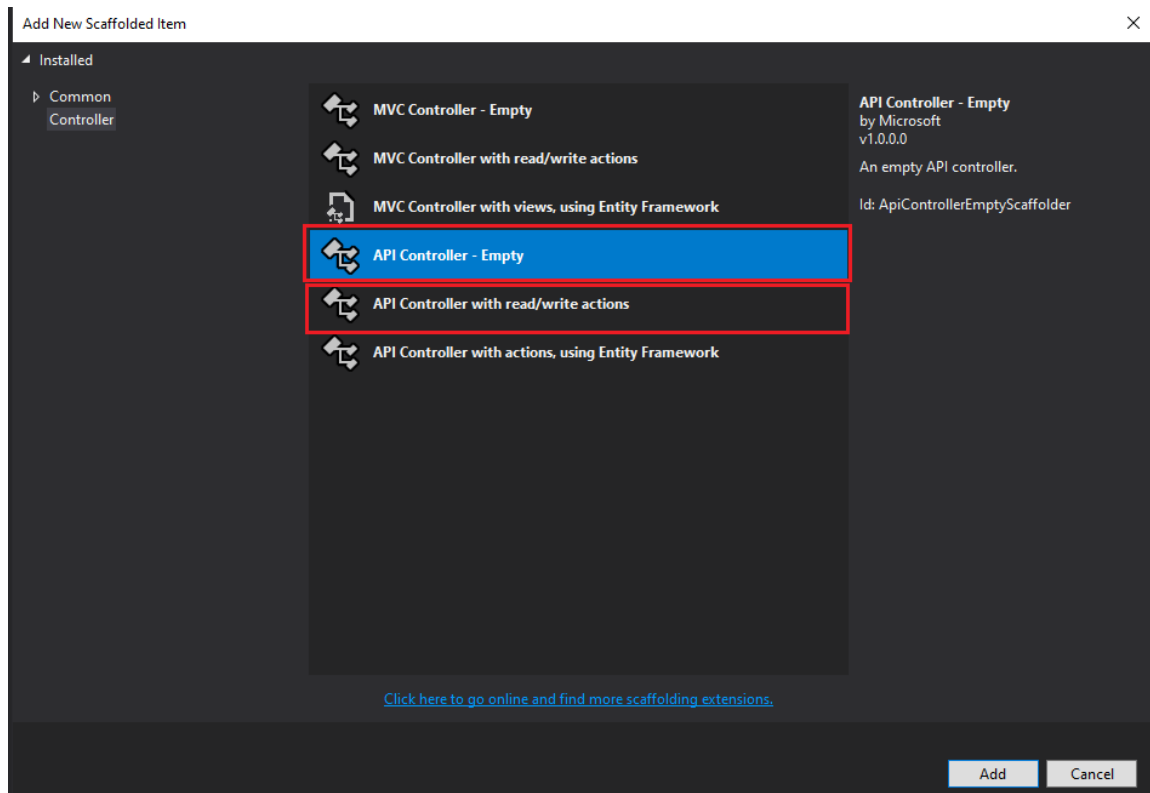
ControllerBase provee muchas properties y métodos que son útiles para manejar requests HTTP. Por ejemplo **CreatedAction** retorna el código de error **201**, **BadRequest** retorna **400**, **NotFound** retorna **404**, entre otros. Para mas información sobre los diferentes métodos y properties vean

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase?view=aspnetcore-3.1>

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. Veamos de crear un endpoint para traer todos las películas que están cargadas en el sistema 🍷. Antes de lanzarnos a escribir **verbos HTTP** y unos cuantos métodos, veamos antes que son los **attributes**.

Attributes

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

[ApiController] attribute

Este 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 VidlyControllerBase : ControllerBase
{
}
```

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

Otra opción que podemos optar es que al estar usando la version 3.1 de [ASP.NET Core](#) podemos 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

<u>Aa</u> Verbo	≡ Attribute	≡ Property
<u>GET</u>	[HttpGet] o [HttpGet(string template)]	Identifica una acción que soporta el verbo GET de HTTP

Aa Verbo	≡ Attribute	≡ Property
<u>POST</u>	[HttpPost] o [HttpPost(string template)]	Identifica una acción que soporta el verbo POST de HTTP
<u>PUT</u>	[HttpPut] o [HttpPut(string template)]	Identifica una acción que soporta el verbo PUT de HTTP
<u>DELETE</u>	[HttpDelete] o [HttpDelete(string template)]	Identifica una acción que soporta el verbo DELETE de HTTP
<u>Untitled</u>	ASD	

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. También podemos encontrar algunas properties como por ejemplo **Name**, **Order**, **Template** y **HttpMethods**; estas properties son accedidas por cualquier verbo HTTP porque todos heredan de **HTTPMethodAttribute**.

La property **Name** la podemos usar para hacer re-direccionamiento de un método a otro a través del nombre, se usaría de la siguiente manera: **[HttpGet("template",Name="Nombre")]** el parámetro **template** puede ir como no.

Veamos de crear varios endpoints, para esto solo se mostrar 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();
}
```

Acá vimos las diferentes formas de crear verbos HTTP con sus respectivos attributes. Si se fijan también usamos 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:

Model binding

<u>Aa</u> Attribute	<u>≡</u> Ubicacion de la info
<u>[FromBody]</u>	En el body de la request
<u>[FromHeader]</u>	En el header de la request
<u>[FromQuery]</u>	En los parametros de la request
<u>[FromRoute]</u>	En la misma request
<u>Untitled</u>	

⚠ Cuidado

- No utilizar **[FromRoute]** cuando el valor contiene el carácter / porque se tomara lo siguiente como un nuevo nivel, en vez usar **[FromQuery]**
- Si se espera que un valor obligatoriamente se tenga que pasar por la request se tiene que especificar **[FromRoute]**, en caso de no hacerlo se puede pasar el valor como un query param.

Para la firma:

```
[HttpGet]
public IActionResult Get(int id)
{
    return Ok();
}
```

Se puede asociar los siguientes endpoints:

1. *api/movies/1*

2. `api/movies?id=1`

Seguramente ahora se están preguntando como es que se agarran los valores de la request con los tipos en los parámetros. La respuesta a su pregunta es gracias al mecanismo de **Model Binding** que nos provee **ASP.NET Core**.

Model Binding es la automatización del proceso de convertir el string que escriben los clientes que son las requests 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 propiedades publicas.

Para saber mas sobre **Model Binding** vean lo siguiente:

Model Binding in ASP.NET Core

This article explains what model binding is, how it works, and how to customize its behavior. View or download sample code (how to download). Controllers and Razor pages work with data that comes

 <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.1>



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:

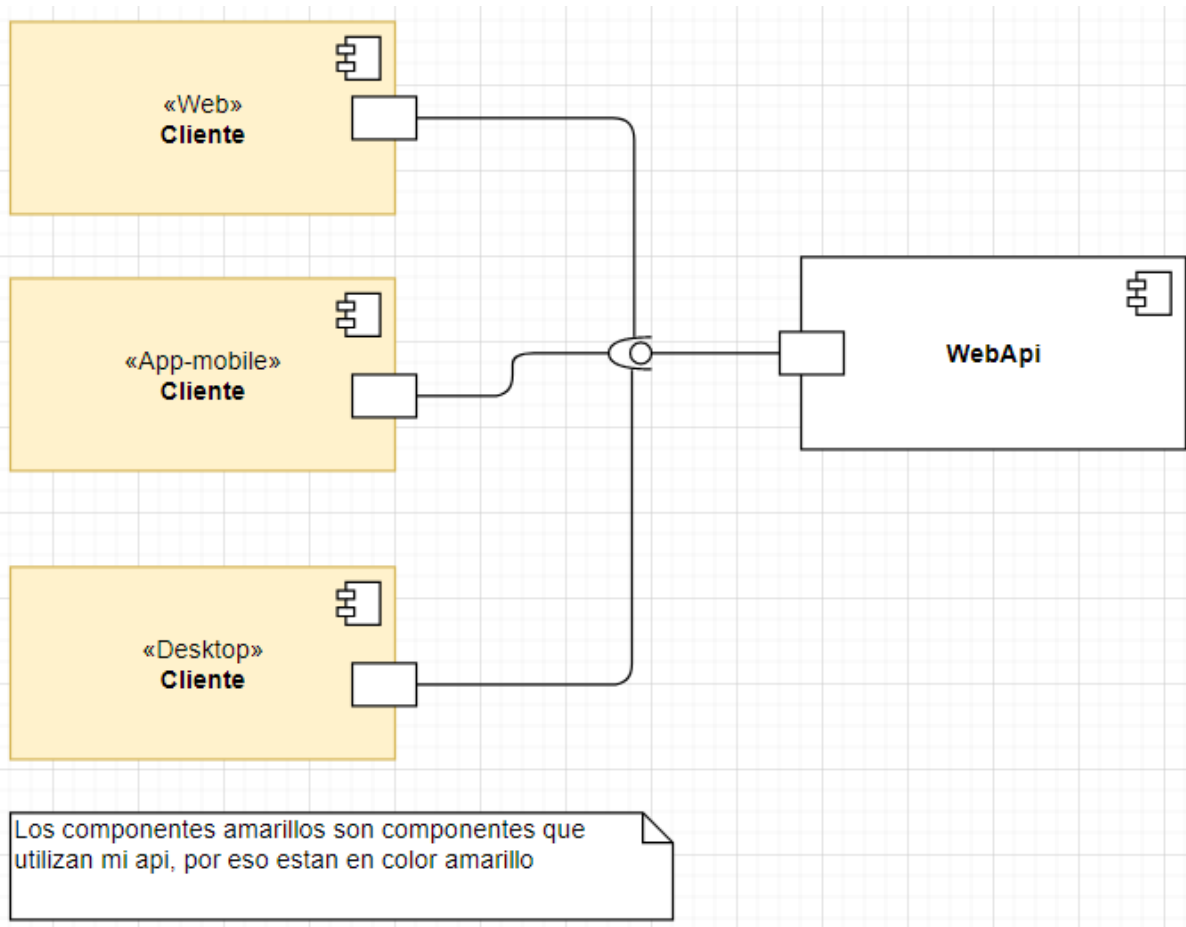
- Prevenir mostrar de mas
- Esconder properties a clientes que no se suponen que vean
- Omitir algunas propiedades para reducir tamaño
- Minimizar las relaciones entre objetos.

- Minimizar el impacto de cambio a los clientes

Cual es el objetivo de los modelos?

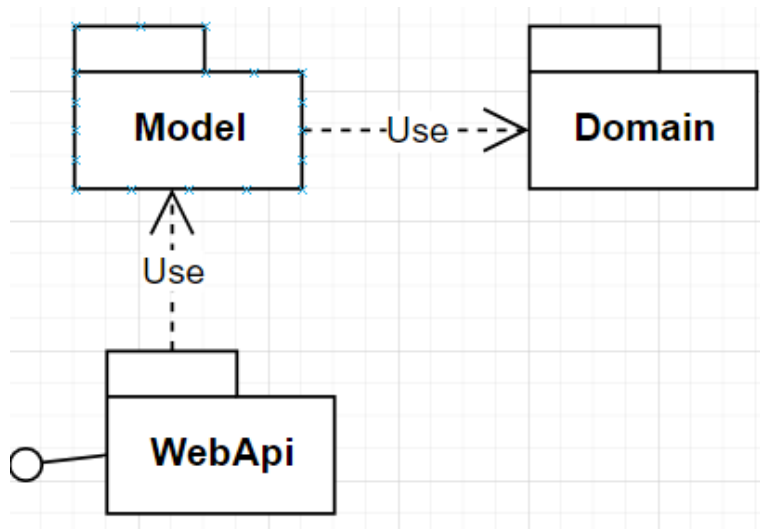
Como antes habíamos visto que la **api** era la implementación del **patron Fachada**, la realización de **modelos** o **dto** es la implementación del **patron Adapter**. De esta forma estamos **envolviendo** las entidades para que los diferentes clientes no se vean afectados si las entidades cambian 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.

Bibliografia

- **Create web APIs with ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-3.1#apicontroller-attribute>
- **Model Binding in ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.1>
- **Tutorial: Create a web API with ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio-code>