ASP.NET Core

Conocimientos

- Mínimos:
  - HTML, CSS, DOM, XML
  - JavaScript
  - C#
  - ADO.NET
- Deseables
  - ASP.NET
  - Entity Framework
  - LINQ
  - JSON, AJAX
  - JQuery,
  - Patrones:
    - MVC, MVP, MVVM
    - DI, IoC
  - Nuget

Contenidos

- Diferencias arquitectónicas
  - .NET Framework y .NET Core
  - ASP.NET MVC y ASP.NET Core
  - Diferencias de hospedaje, inicio de la aplicación
  - Proporcionar archivos estáticos
- Proyectos
  - Estructura del proyecto
  - Configuración
  - Enrutamiento
  - Registro
- Inyección de dependencias
  - Métodos de registro del servicio
  - Duraciones de servicios
  - Inserción de dependencias
  - Servicios proporcionados
- Middleware de ASP.NET Core
  - Creación de Filtros
  - Orden del middleware
  - Middleware integrado
- Modelos
  - Entity Framework Core
- Controladores
  - Controller y ApiController
  - Acciones síncronas y asíncronas
  - Negociación de contenido
  - Enrutar a acciones
- Vistas
  - Asistentes de etiquetas (tag helpers)
  - Componentes de vista (view component)
  - Razor Pages en ASP.NET Core
  - Validación de Modelos
- Desarrollo del lado del cliente
  - Aplicaciones de una sola página
  - Agrupar y minimizar
- Seguridad de una Aplicación Web
  - Permisos basados en Claims
  - Permisos basados en Políticas
- Depuración, Pruebas Unitarias y Refactorización
  - Refactorizando Código y Depuración
  - Pruebas Unitarias
  - Procesando Excepciones no controladas
  - Desarrollo Orientado a Test (TDD)
- Despliegue y puesta en producción de Aplicaciones Web
  - Hospedaje e implementación
  - Proceso de Despliegue para producción

Creación de proyectos

ARQUITECTURA DE UNA APLICACIÓN MVC

Tipos de Proyectos

- Aplicaciones web (Razor Pages)
  - Razor Pages facilita la programación de escenarios centrados en páginas, que no necesitan la complejidad de controladores y vistas.
- Aplicaciones web (Modelo-Vista-Controlador)
  - ASP.NET Core MVC es un completo marco de trabajo para compilar aplicaciones web y API mediante el patrón de diseño Modelo-Vista-Controlador.
- API
  - ASP.NET Core admite la creación de servicios RESTful, lo que también se conoce como API Web
- SPA
  - ASP.NET Core MVC con ClientApp en Angular, React, React con Redux
- Blazor
  - Es una plataforma de trabajo para la creación de interfaces de usuario web interactivas del lado cliente con .NET usando WebAssembly

Estructura de un proyecto MVC

- wwwroot: Contiene recursos estáticos, como imágenes o archivos HTML, JavaScript y CSS.
- Controllers: Contiene las clases con los controladores que responden a la entrada desde el navegador, decidir qué hacer con él y devolver la respuesta al usuario.
- Models: Contiene las clases del modelo.
- Data: Contiene las clases del EF para el mantenimiento del contexto.
- Views: Contiene las plantillas de interfaz de usuario, estructuradas en subcarpetas por controlador.
- Views/Shared: Contiene las plantillas y elementos compartidos.
- Areas: Contienen las carpetas de las áreas y sus subcarpetas (Controllers, Views, Models).
- Program.cs: Contiene un método Main, el punto de entrada administrado de la aplicación.
- Startup.cs: Configura el comportamiento de la aplicación, como el enrutamiento entre páginas.
- appsettings.json: Fichero de configuración externa, dispone de versiones appsettings.Production.json o appsettings.Development.json

Host genérico de .NET en ASP.NET Core

- El host es un objeto que encapsula todos los recursos de la aplicación para la administración y el control sobre el inicio de la aplicación y el apagado estable.
- Normalmente se configura, compila y ejecuta el host por el código de la clase Program. El método Main realiza las acciones siguientes:
  - Llama a un método CreateHostBuilder para crear y configurar un objeto del generador.
  - Llama a los métodos Build y Run en el objeto del generador.
- Las aplicaciones de ASP.NET Core configuran e inician un host. El host es responsable de la administración del inicio y la duración de la aplicación. Como mínimo, el host configura un servidor y una canalización de procesamiento de solicitudes. El host también puede configurar el registro, la inserción de dependencias y la configuración. Establece el servidor Kestrel como servidor web y lo configura por medio de los proveedores de configuración de hospedaje de la aplicación.
- El servidor Kestrel es la implementación de servidor HTTP multiplataforma predeterminada. Kestrel proporciona el mejor rendimiento y uso de memoria, pero carece de algunas de las características avanzadas.

Clase Startup

- Las aplicaciones de ASP.NET Core utilizan una clase para configurar los servicios y la canalización de solicitudes de la aplicación, que se denomina Startup por convención:
  - Incluye opcionalmente un método ConfigureServices para configurar los servicios de la aplicación. Un servicio es un componente reutilizable que proporciona funcionalidades de la aplicación. Los servicios se registran en ConfigureServices y se usan en la aplicación a través de la inyección de dependencias (DI) o ApplicationServices.
  - Incluye un método Configure para crear la canalización de procesamiento de solicitudes de la aplicación.
- La clase Startup se especifica cuando se crea el host de la aplicación. Habitualmente, la clase Startup se especifica en el generador de host del main mediante una llamada al método WebHostBuilderExtensions.UseStartup<TStartup>.

Inyección de dependencias (servicios)

- ASP.NET Core incluye un marco de inyección de dependencias (DI) integrado que pone a disposición los servicios configurados a lo largo de una aplicación. Por ejemplo, un componente de registro es un servicio.
- Se agrega al método Startup.ConfigureServices el código para configurar (o registrar) servicios. Por ejemplo:

```
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<MovieContext>(options =>
     options.UseSqlServer(Configuration.GetConnectionString("MovieContext")));
```

- Los servicios se suelen resolver desde la inyección de dependencias mediante la inyección de constructores. Con la inyección de constructores, una clase declara un parámetro de constructor del tipo requerido o una interfaz. El marco de inyección de dependencias proporciona una instancia de este servicio en tiempo de ejecución.

```
public class IndexModel : PageModel {
    private readonly MovieContext _context;
    public IndexModel(MovieContext context)  { _context = context; }
```

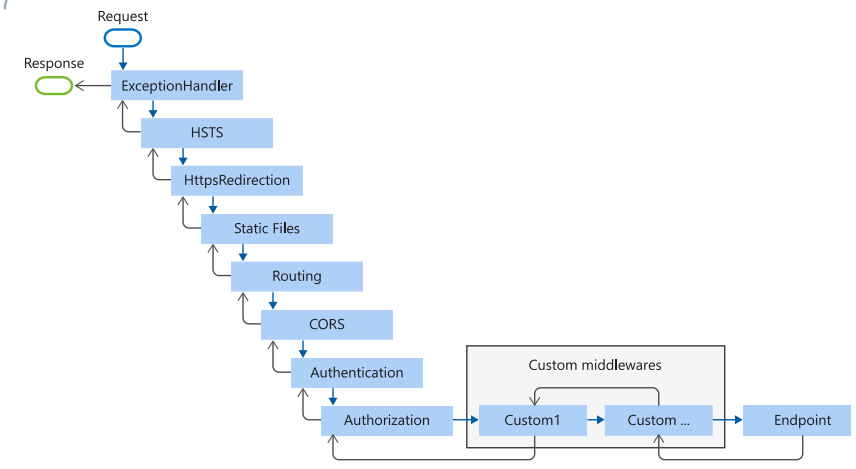

Middleware

- La canalización de control de solicitudes se compone de una serie de componentes de software intermedio.
- Cada componente lleva a cabo las operaciones en un contexto HttpContext e invoca el middleware siguiente de la canalización, o bien finaliza la solicitud.
- Normalmente, se agrega un componente de software intermedio a la canalización al invocar un método de extensión Use... en el método Startup.Configure.

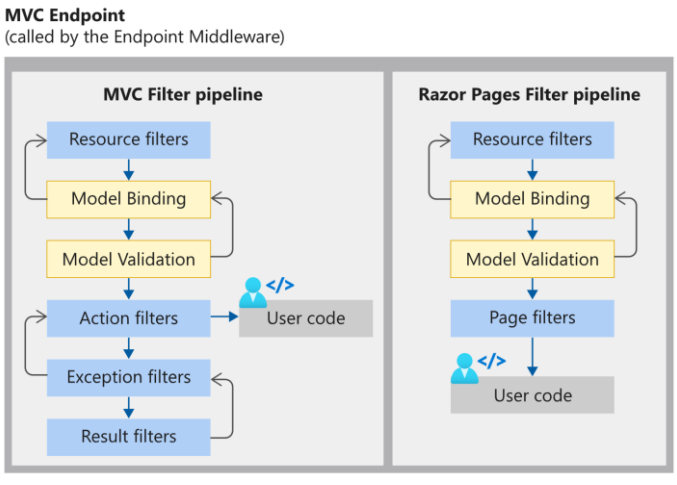
```
public void Configure(IApplicationBuilder app) {
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
```

- ASP.NET Core incluye un amplio conjunto de middleware integrado. También se pueden escribir componentes de middleware personalizados.

Orden del middleware



Punto de conexión



Configuración y entornos

- ASP.NET Core proporciona un marco de configuración que obtiene la configuración como pares nombre/valor de un conjunto ordenado de proveedores de configuración. Hay disponibles proveedores de configuración integrados para una gran variedad de orígenes, como archivos .json y .xml, variables de entorno y argumentos de línea de comandos. De manera predeterminada, las aplicaciones de ASP.NET Core están configuradas para leer desde appsettings.json, variables de entorno, la línea de comandos, etc.
- Los entornos de ejecución, como Development, Staging y Production, son un concepto de primera clase en ASP.NET Core. Especificando el entorno que ejecuta una aplicación en la variable de entorno ASPNETCORE_ENVIRONMENT, ASP.NET Core lee dicha variable de entorno al inicio de la aplicación y almacena el valor en una implementación IWebHostEnvironment. Esta implementación está disponible en cualquier parte de una aplicación a través de la inyección de dependencias (DI).

appsettings.json

- El elemento JsonConfigurationProvider predeterminado carga la configuración en el siguiente orden:
  - appsettings.json
  - appsettings.*Environment*.json: Los valores de appsettings.*Environment*.json invalidan las claves de appsettings.json.
- La mejor manera de leer valores de configuración relacionados es usar el patrón de opciones o acceder directamente a las claves.

```
"Position": { "Title": "Editor", "Name": "Joe Smith" }
```

```
public class PositionOptions {
    public const string Position = "Position";
    public string Title { get; set; }
    public string Name { get; set; }
}
```

```
public Startup(IConfiguration configuration) {  Configuration = configuration; }
public IConfiguration Configuration { get; }
```

```
var positionOptions = new PositionOptions();
Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);
string userName = Configuration.GetSection("Position")["Name"];
```

Registro y Control de errores

- ASP.NET Core es compatible con una API de registro que funciona con una gran variedad de proveedores de registro integrados y de terceros. Los proveedores disponibles incluyen:
  - Consola
  - Depuración
  - Seguimiento de eventos en Windows
  - Registro de errores de Windows
  - TraceSource
  - Azure App Service
  - Azure Application Insights
- ASP.NET Core tiene características integradas para controlar los errores, tales como:
  - Una página de excepciones para el desarrollador
  - Páginas de errores personalizados
  - Páginas de códigos de estado estáticos
  - Control de excepciones de inicio

Enrutamiento

- Una ruta es un patrón de dirección URL que se asigna a un controlador. El controlador normalmente es una página de Razor, un método de acción en un controlador MVC o un software intermedio. El enrutamiento de ASP.NET Core permite controlar las direcciones URL usadas por la aplicación.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    app.UseRouting();
    app.UseEndpoints(endpoints => {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
```

- La raíz web es la ruta de acceso base para los archivos de recursos estáticos públicos: Hojas de estilo (.css), JavaScript (.js), Imágenes (.png, .jpg), …
- De manera predeterminada, los archivos estáticos solo se sirven desde el directorio raíz web y sus subdirectorios. La ruta de acceso raíz web se establece de manera predeterminada en {raíz del contenido}/wwwroot.
- En los archivos Razor (.cshtml), la virgulilla (~/) apunta a la raíz web. Una ruta de acceso que empieza por ~/ se conoce como ruta de acceso virtual.

Modelo de hospedaje mínimo (v.6)

- Reducen considerablemente la cantidad de archivos y líneas de código que se necesitan para crear una aplicación. Por ejemplo, la aplicación web ASP.NET Core vacía crea un archivo de C# con cuatro líneas de código y es una aplicación completa.
- Unifican Startup.cs y Program.cs en un archivo Program.cs único.
- Utilizan instrucciones de nivel superior a fin de minimizar el código necesario para una aplicación.
- Utilizan directivas using globales a fin de eliminar o minimizar la cantidad de líneas de instrucción using que se requieren.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

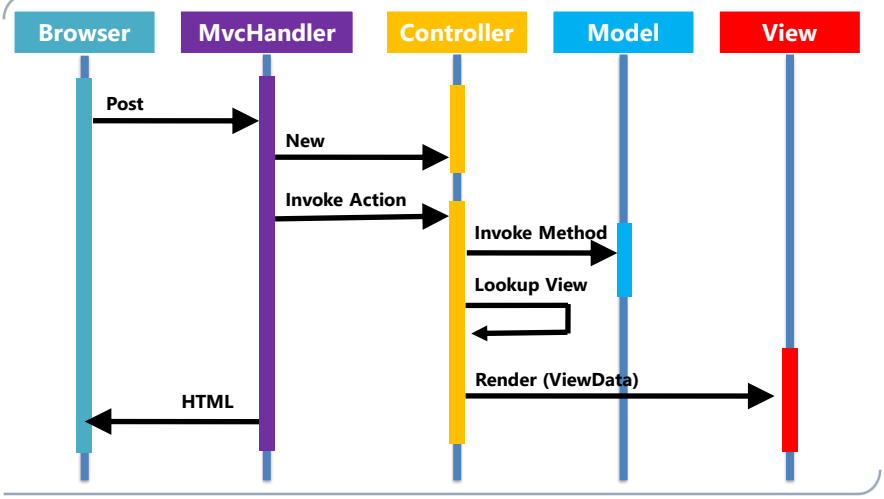

Estructura de un proyecto MVC 6

- wwwroot: Contiene recursos estáticos, como imágenes o archivos HTML, JavaScript y CSS.
- Controllers: Contiene las clases con los controladores que responden a la entrada desde el navegador, decidir qué hacer con él y devolver la respuesta al usuario.
- Models: Contiene las clases del modelo.
- Data: Contiene las clases del EF para el mantenimiento del contexto.
- Views: Contiene las plantillas de interfaz de usuario, estructuradas en subcarpetas por controlador.
- Views/Shared: Contiene las plantillas y elementos compartidos.
- Areas: Contienen las carpetas de las áreas y sus subcarpetas (Controllers, Views, Models).
- Program.cs: instrucciones de nivel superior, el punto de entrada administrado de la aplicación y la configuración del comportamiento de la aplicación, como el enrutamiento entre páginas.
- appsettings.json: Fichero de configuración externa, dispone de versiones appsettings.Production.json o appsettings.Development.json

Fases de ejecución de MVC

1. Recibir la primera solicitud para la aplicación
2. Realizar el enrutamiento
3. Crear el controlador de solicitudes de MVC
4. Crear el controlador
5. Ejecutar el controlador
6. Invocar la acción
7. Ejecutar el resultado
8. Enviar el resultado

El framework en funcionamiento



Principales resultados de la acción

Resultado	Descripción
ViewResult	Representa una vista como una página web.
PartialViewResult	Representa una vista parcial, que define una sección de una vista que se puede representar dentro de otra vista.
RedirectResult	Redirecciona a otro método de acción utilizando su URL.
RedirectToRouteResult	Redirecciona a otro método de acción.
ContentResult	Devuelve un tipo de contenido definido por el usuario.
JsonResult	Devuelve un objeto JSON serializado.
FileResult	Devuelve la salida binaria para escribir en la respuesta.
EmptyResult	Representa un valor devuelto que se utiliza si el método de acción debe devolver un resultado null (vacío).
StatusCodeResult	Proporciona un modo para devolver un resultado de la acción con un código de estado de respuesta HTTP.

NuGet Package Manager

- Gestor de paquetes para desarrolladores
- Se instala como extensión
- Cuenta con una amplia base de datos actualizada de paquetes
- Simplifica el uso de componentes externos.
  - Localización
  - Descarga (¡con dependencias!)
  - Instalación / desinstalación
  - Configuración
  - Actualización
- Dispone de la consola de administración de paquetes.

Ampliación

MODELOS

Code First

- Instalación de Entity Framework Core
  - Install-Package Microsoft.EntityFrameworkCore.SqlServer
  - Install-Package Microsoft.EntityFrameworkCore.Tools
- Creación de las entidades (modelos) en la carpeta Models (nombres en singular)
- Creación de una clase de contexto de base de datos (hereda de DbContext o uno de sus herederos) en la carpeta Data, agregando un DbSet por entidad (nombres en plural)

```
public class ApplicationDbContext : IdentityDbContext {
      public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
         : base(options) {
      }
      public DbSet<Producto> Productos { get; set; }
   }
```

- Generar y revisar el archivo de migración:

```
Add-Migration InitialCreate
```

- Actualizar (o crear) la base de datos

```
Update-Database
```

Database First

- Instalación de Entity Framework Core
  - Install-Package Microsoft.EntityFrameworkCore.SqlServer
  - Install-Package Microsoft.EntityFrameworkCore.Tools
- La ingeniería inversa: Database First
  - get-help scaffold-dbcontext –detailed
- Generar clases:
  - Scaffold-DbContext "Data Source=.;Initial Catalog=AdventureWorksLT2017;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False" Microsoft.EntityFrameworkCore.SqlServer -DataAnnotations -ContextDir Infrastructure.Data.UnitOfWork -Context TiendaDbContext -OutputDir Domain.Entities -Schemas SalesLT

Database First

- La cadena de conexión en appsettings.json:

```
{
  "ConnectionStrings": {
    "TiendaConnection": "Data Source=.;Initial Catalog=AdventureWorksLT2017;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False"
  },
}
```

- La inyección de dependencias en Startup.ConfigureServices:

```
services.AddDbContext<TiendaDbContext>(options => options.UseSqlServer(
      Configuration.GetConnectionString("TiendaConnection")));
builder.Services.AddDbContext<TiendaDbContext>(options =>
  options.UseSqlServer(
    builder.Configuration.GetConnectionString("TiendaConnection")));
```

- Eliminar el método OnConfiguring con la cadena de conexión de la clase TiendaDbContext.

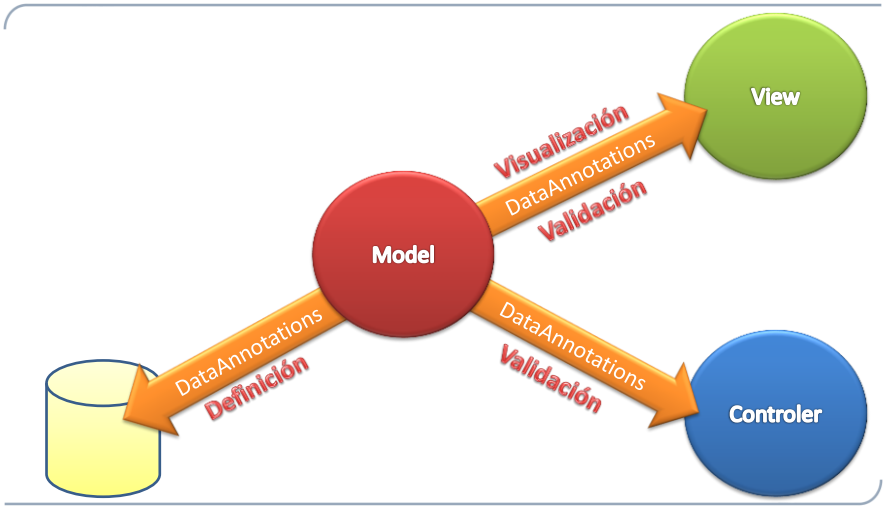
Database First

- Instalación de las herramientas
  - dotnet tool install --global dotnet-ef
- Comprobar la instalación:
  - dotnet ef
- Instalación de Entity Framework Core
  - Install-Package Microsoft.EntityFrameworkCore.SqlServer
  - Install-Package Microsoft.EntityFrameworkCore.Tools
- Generar clases:
  - dotnet ef dbcontext scaffold "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Curso" Microsoft.EntityFrameworkCore.SqlServer

DataAnnotations

- System.ComponentModel.DataAnnotations;
- Permiten definir los metadatos de los modelos de datos para su uso por entidades externas.
- Principio DRY: No repetir
- Cuenta con decoradores para:
  - Definición del modelo de datos: claves (PK), asociaciones (FK), simultaneidad, …
  - Interfaz de usuario y localización: Título, descripción, formato, solo lectura, …
  - Validaciones y errores personalizados: Obligatoriedad, longitudes, formatos, …

Contexto Declarativo



Visualización

Decorador	Descripción
DataType	Especifica el nombre de un tipo adicional que debe asociarse a un campo de datos.
Display	Permite especificar las cadenas traducibles de los tipos y miembros de las clases parciales de entidad.
DisplayFormat	Especifica el modo en que los datos dinámicos de ASP.NET muestran y dan formato a los campos de datos.
ScaffoldColumn	Especifica si una clase o columna de datos usa la técnica scaffolding.
ScaffoldTable	Especifica si una clase o tabla de datos usa la técnica scaffolding.
UIHint	Especifica la plantilla o el control de usuario que los datos dinámicos usan para mostrar un campo de datos.
HiddenInput	(Microsoft.AspNetCore.Mvc) Indica que una propiedad se debería presentar como un elemento input oculto.

[Display]

- **Name**: valor que se usa para mostrarlo en la interfaz de usuario (Título, Etiqueta, …).
- **Description**: valor que se usa para mostrar una descripción en la interfaz de usuario.
- **Prompt**: valor que se usará para establecer la marca de agua para los avisos en la interfaz de usuario.
- **ShortName**: valor que se usa para la etiqueta de columna de la cuadrícula.
- **GroupName**: valor que se usa para agrupar campos en la interfaz de usuario.
- **Order**: número del orden de la columna.

Tipos asociados

Asociado	Descripción
DateTime	Representa un instante de tiempo, expresado en forma de fecha y hora del día.
Date	Representa un valor de fecha.
Time	Representa un valor de hora.
Duration	Representa una cantidad de tiempo continua durante la que existe un objeto.
PhoneNumber	Representa un valor de número de teléfono.
Currency	Representa un valor de divisa.
Text	Representa texto que se muestra.
Html	Representa un archivo HTML.
MultilineText	Representa texto multilínea.
EmailAddress	Representa una dirección de correo electrónico.
Password	Represente un valor de contraseña.
Url	Representa un valor de dirección URL.
ImageUrl	Representa una URL en una imagen.
CreditCard	Representa un número de tarjeta de crédito.
PostalCode	Representa un código postal.
Upload	Representa el tipo de datos de la carga de archivos.

Validación

Decorador	Descripción
Required	Especifica que un campo de datos necesita un valor.
DataType	Especifica el nombre de un tipo adicional que debe asociarse a un campo de datos. Decoradores especializados: CreditCard, EmailAddress, EnumDataType, FileExtensions, Phone, Url
Compare	Compara dos propiedades.
Range	Especifica las restricciones de intervalo numérico para el valor de un campo de datos.
StringLength	Especifica la longitud mínima y máxima de caracteres que se permiten en un campo de datos. Decoradores especializados: MinLength, MaxLength
RegularExpression	Especifica que un valor de campo de datos debe coincidir con la expresión regular especificada.
CustomValidation	Especifica un método de validación personalizado que se utiliza para validar una propiedad o una instancia de clase.

Validación

Decorador	Descripción
CreditCard	Especifica que el valor de un campo de datos es un número de tarjeta de crédito. Requiere métodos adicionales de validación de jQuery.
EmailAddress	Valida que la propiedad tiene un formato de correo electrónico.
Phone	Especifica que un valor de campo de datos es un número de teléfono con formato correcto.
Url	Valida que la propiedad tiene un formato de dirección URL.
Remote	Valida la entrada en el cliente mediante una llamada a un método de acción en el servidor. Está en el espacio de nombres Microsoft.AspNetCore.Mvc.
ValidateNever	Indica que una propiedad o parámetro debe excluirse de la validación. Está en el espacio de nombres Microsoft.AspNetCore.Mvc.ModelBinding.Validation.

Validación manual de objetos

- En System.ComponentModel.DataAnnotations, la clase estática Validator ofrece métodos que permiten realizar las comprobaciones de forma directa sobre objetos o propiedades concretas.

```
IEnumerable<ValidationResult> getValidationErrors(object obj)    {
    var validationResults = new List<ValidationResult>();
    var context = new ValidationContext(obj, null, null);
    Validator.TryValidateObject(obj,
            context,
            validationResults,
            true);
    return validationResults;
}
```


IValidatableObject

- Obliga a implementar un único método, llamado Validate(), que determina si el objeto especificado es válido.
- Será invocado automáticamente por TryValidateObject() siempre que no encuentre errores al comprobar las restricciones especificadas mediante anotaciones.
- Devolverá una lista de objetos ValidationResult con los resultados de las comprobaciones.
- En las clases prescriptivas (EF) se aplica con una partial class.

```
public partial class Persona : IValidatableObject {
    public IEnumerable<ValidationResult> Validate(ValidationContext
        validationContext) {
        if (FechaNacimiento.Date.CompareTo(DateTime.Today) > 0)
            yield return new ValidationResult("Todavía no ha nacido", new[]
                { nameof(FechaNacimiento) });
```

- Interfaces relacionados: IDataErrorInfo, INotifyDataErrorInfo

Anotar clases ya existentes

- Se requiere una clase auxiliar con las propiedades a decorar decoradas.
- Declarativa: [MetadataType(typeof(tipo))]
  - Se aplica con una partial class (en el mismo ensamblado que la clase a anotar).
- Imperativa: System.ComponentModel.TypeDescriptor y AssociatedMetadataTypeTypeDescriptionProvider:

```
var descriptionProvider = new
    AssociatedMetadataTypeTypeDescriptionProvider(
      typeof(Friend),  typeof(FriendMetadata));
TypeDescriptor.AddProviderTransparent(
    descriptionProvider, typeof(Friend));
```


Validación del lado cliente

- La validación del lado cliente impide realizar el envío hasta que el formulario sea válido, el botón Enviar ejecuta JavaScript para enviar el formulario o mostrar mensajes de error, evitando un recorrido de ida y vuelta innecesario en el servidor cuando hay errores de entrada en un formulario.
- El script de validación discreta de jQuery es una biblioteca de front-end personalizada de Microsoft que se basa en el conocido complemento de validación de jQuery. Los asistentes de etiquetas y los HTML Helper usan los atributos de validación y escriben metadatos de las propiedades del modelo para representar atributos data- de HTML 5 para los elementos de formulario que necesitan validación.
- Se deben activar las siguientes referencias a script:
  - _Layout.cshtml

```
<script src="~/lib/jquery/dist/jquery.min.js"></script>
```

  - _ValidationScriptsPartial.cshtml

```
<script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
</script>
```

Atributo [Remote]

- Implementa la validación del lado cliente que requiere llamar a un método en el servidor para determinar si la entrada del campo es válida. Por ejemplo, requiere acceso a datos.
- La validación remota se realiza mediante un método de acción para que lo invoque de jQuery que espera una respuesta JSON:
  - true significa que los datos de entrada son válidos.
  - false, undefined o null significan que la entrada no es válida y muestra el mensaje de error predeterminado, o cualquier otra cadena y muestra la cadena como un mensaje de error personalizado.

```
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyEmail(string email) {
   return _userService.isUnique(email) ? Json(true) : Json($"Email {email} is already in use.");
}
```

- En la clase de modelo, se anota la propiedad con un atributo [Remote] apuntando al método de acción de validación:

```
[Remote(action: "VerifyEmail", controller: "Validations")]
public string Email { get; set; }
```

- La propiedad AdditionalFields del atributo [Remote] permite validar combinaciones de campos con los datos del servidor.

Atributos personalizados

```
public class NIFAttribute : ValidationAttribute {
   public NIFAttribute() : this("No es un NIF válido.") { }
   public NIFAttribute(Func<string> errorMessageAccessor) : base(errorMessageAccessor) { }
   public NIFAttribute(string errorMessage) : base(errorMessage) { }
   public string DefaultErrorMessage => ErrorMessageString;
   protected override ValidationResult IsValid(object value, ValidationContext validationContext) {
      if (value == null) return ValidationResult.Success;
      if (value is String cad) {
         cad = cad.ToUpper();
         if (Regex.IsMatch(cad, @"^\d{2,8}[A-Z]$") &&
            cad[^1] == "TRWAGMYFPDXBNJZSQVHLCKE"[(int)(long.Parse(cad[0..^1]) % 23)])
            return ValidationResult.Success;
      }
      return new ValidationResult(ErrorMessageString);
   }
}

[NIF]
public string NIF { get; set; }
```

Validación personalizada en cliente

```
using Microsoft.AspNetCore.Mvc.DataAnnotations;
public class NIFAttributeAdapter : AttributeAdapterBase<NIFAttribute> {
   public NIFAttributeAdapter(NIFAttribute attribute, IStringLocalizer stringLocalizer)
      : base(attribute, stringLocalizer) { }
   public override void AddValidation(ClientModelValidationContext context) {
      MergeAttribute(context.Attributes, "data-val", "true");
      MergeAttribute(context.Attributes, "data-val-nif", GetErrorMessage(context));
   }
   public override string GetErrorMessage(ModelValidationContextBase validationContext) =>
      Attribute.DefaultErrorMessage;
}
public class CustomValidationAttributeAdapterProvider : IValidationAttributeAdapterProvider {
   private readonly IValidationAttributeAdapterProvider baseProvider = new ValidationAttributeAdapterProvider();
   public IAttributeAdapter GetAttributeAdapter(ValidationAttribute attribute,
      IStringLocalizer stringLocalizer) {
      if (attribute is NIFAttribute NIFAttribute)
         return new NIFAttributeAdapter(NIFAttribute, stringLocalizer);
      return baseProvider.GetAttributeAdapter(attribute, stringLocalizer);
   }
}
// Registrar en Startup.ConfigureServices
services.AddSingleton<IValidationAttributeAdapterProvider, CustomValidationAttributeAdapterProvider>();
```

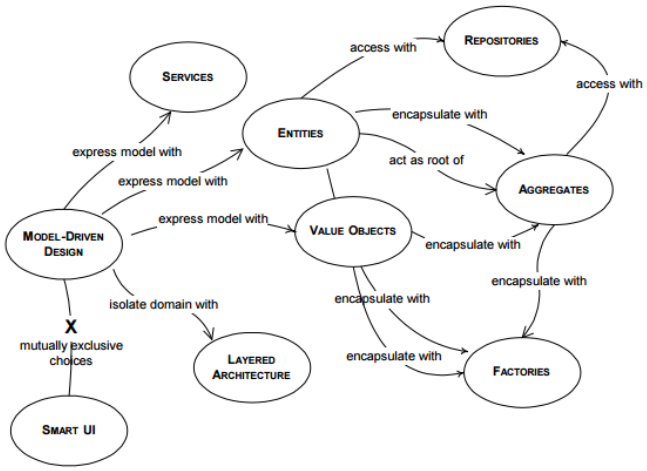

Validación personalizada en cliente

```
// Añadir fichero wwwroot\js\validadores.js
$.validator.addMethod('nif', function (value, element, params) {
   var nif = value;

   if (!/^\d{1,8}[A-Za-z]$/.test(nif))
      return false;
   const letterValue = nif.substr(nif.length - 1);
   const numberValue = nif.substr(0, nif.length - 1);
   return letterValue.toUpperCase() === 'TRWAGMYFPDXBNJZSQVHLCKE'.charAt(numberValue % 23);

});
$.validator.unobtrusive.adapters.add('nif', ['year'], function (options) {
   options.rules['nif'] = {};
   options.messages['nif'] = options.message;
});

// Añadir en Views\Shared\_ValidationScriptsPartial.cshtml
<script src="~/js/validadores.js"></script>
```

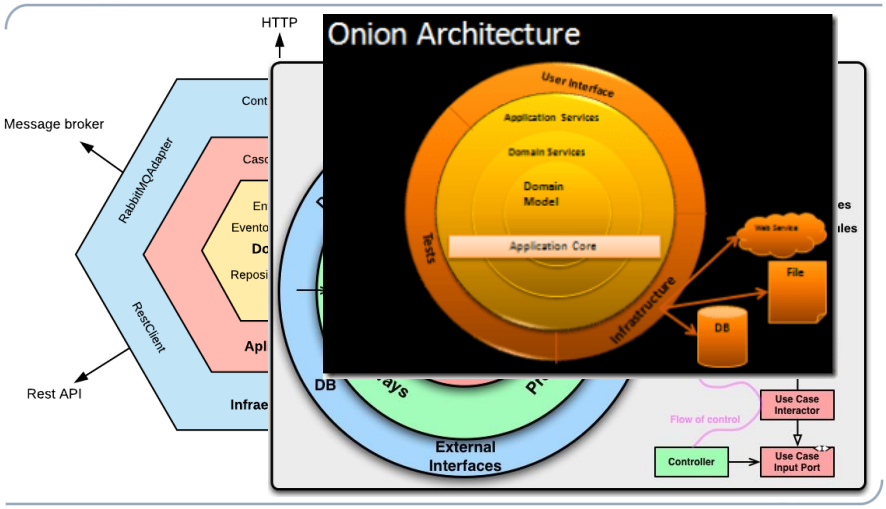
ModelView

- Los modelos representan entidades de dominio (capa de dominio), que pueden requerir ampliaciones para su uso en capa de presentación.
- Los modelos de la capa de presentación se denominan comúnmente como ModelViews, modelos de las vistas.
- No se deben confundir con los ViewModels del patrón MVVM.
- Los ModelViews son clases (con el sufijo ModelView) que suelen encapsular al modelo de dominio (entidad) a través de una propiedad.
- Los controladores pasan los ModelViews a las vistas en sustitución de los modelos.

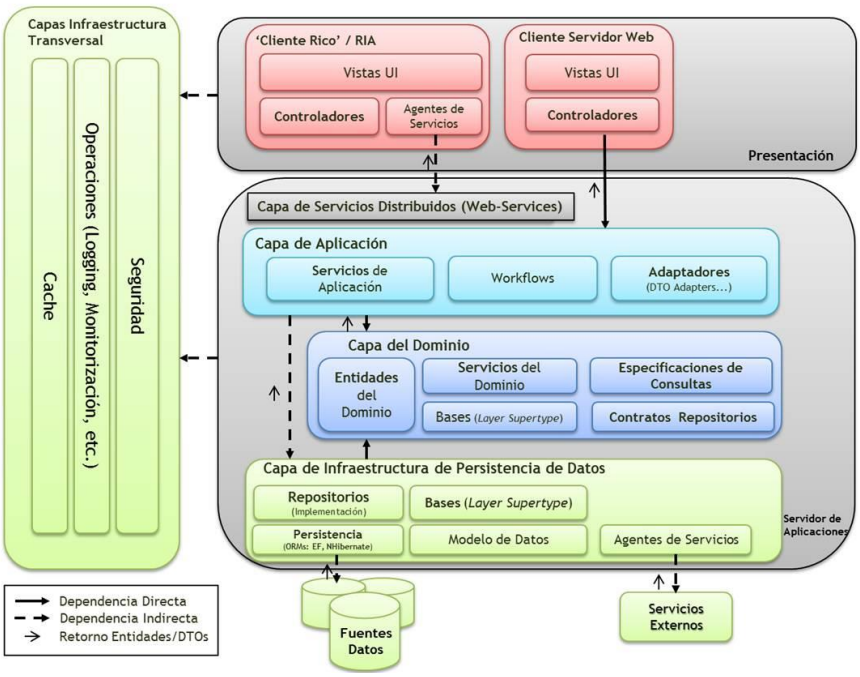
Domain Driven Design

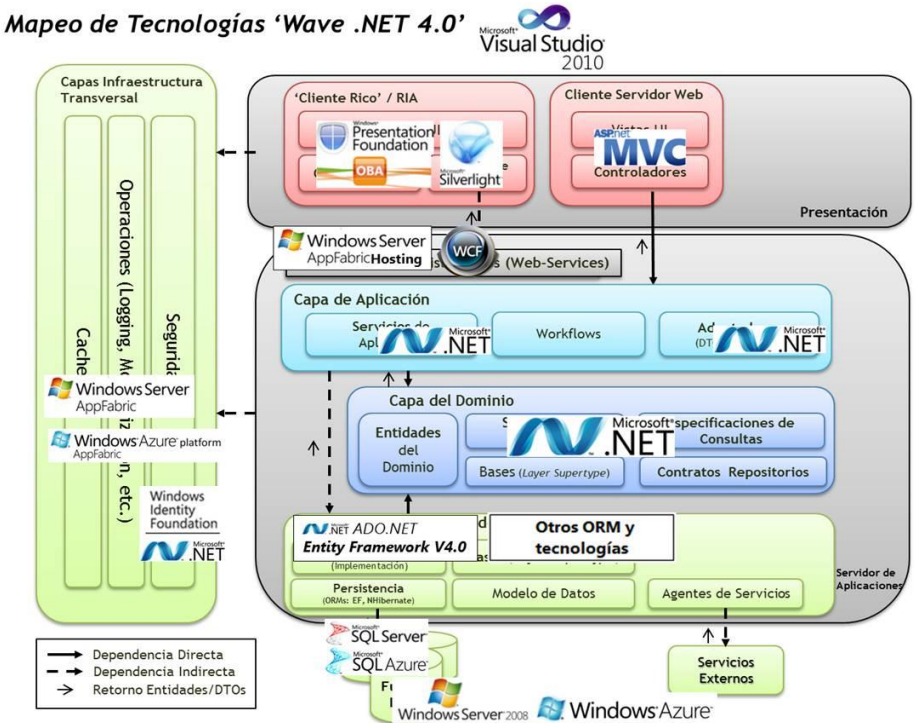


Arquitecturas Concéntricas



Arquitectura N-Capas con Orientación al Dominio





Repositorio

- Un repositorio separa la lógica empresarial de las interacciones con la base de datos subyacente y centra el acceso a datos en un área, lo que facilita su creación y mantenimiento.
- El repositorio pertenecen a la capa de infraestructura y devuelve los objetos del modelo de dominio.
- Deberían implementar el patrón de doble herencia.
- Forma parte de los Domain Driven Design patterns: Domain Entity, Value-Object, Aggregates, Repository, Unit of Work, Specification, Dependency Injection, Inversion of Control (IoC).

Ventajas del Repositorio

- Proporciona un punto de sustitución para las pruebas unitarias. Es fácil probar la lógica empresarial sin una base de datos y otras dependencias externas.
- Las consultas y los modelos de acceso a datos duplicados se pueden quitar y refactorizar en el repositorio.
- Los métodos del controlador pueden usar parámetros fuertemente tipados, lo que significa que el compilador encontrará errores de tipado de datos en cada compilación en lugar de realizar la búsqueda de errores de tipado de datos en tiempo de ejecución durante las pruebas.
- El acceso a datos está centralizado, lo que brinda las siguientes ventajas:
  - Mayor separación de intereses (SoC), uno de los principios de MVC, lo que facilita más el mantenimiento y la legibilidad.
  - Implementación simplificada de un almacenamiento en caché de datos centralizado.
  - Arquitectura más flexible y con menor acoplamiento, que se puede adaptar a medida que el diseño global de la aplicación evoluciona.
- El comportamiento se puede asociar a los datos relacionados (calcular campos, aplicar relaciones, reglas de negocios complejas entre los elementos de datos de una entidad, …).
- Un modelo de dominio se puede aplicar para simplificar una lógica empresarial compleja.

DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red.
- Su finalidad es:
  - Desacoplar del nivel de servicio de la capa de base de datos.
  - Quitar las referencias circulares.
  - Ocultar determinadas propiedades que los clientes no deberían ver.
  - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
  - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más conveniente para los clientes.
  - Evitar el "exceso" y las vulnerabilidades por publicación.

Ampliación

CONTROLADORES

Métodos de acción

- Todos los métodos públicos de la clase son métodos de acción
- Para evitar que un método público sea de acción se decora con [NonAction]
- Pueden tener parámetros.
- Se invocan mediante una URL (GET):
  ~/controlador/método/VPP?NP=VP&…
  - VPP: Valor del parámetro predeterminado (opcional)
  - NP=VP: Pares Nombre Parámetro = Valor Parámetro, opcionales, el primero precedido por ? y el resto por &.

Parámetros de los métodos de acción

- Número de parámetros:
  - Ninguno
  - Uno, predeterminado: Por defecto “id” (definido al mapear la ruta).
    ~/controlador/método/1
  - Uno, no predeterminado: Requiere el par Nombre=Valor.
    ~/controlador/método/?p1=1
  - Varios: el primero puede ser el predeterminado, el resto requieren los pares Nombre=Valor
    ~/controlador/método/1?p2=2&p3=3
    ~/controlador/método/?p1=1&p2=2
- Los valores de los parámetros de método de acción se asignan automáticamente, se mapean por nombre.

Métodos de acción

- Por defecto los parámetros se reciben vía GET, para indicar la recepción vía POST se decoran con [HttpPost].
- El decorador [ActionName] permite asignar un nombre al método de acción distinto al del nombre del método en la clase.
- [ValidateAntiForgeryToken] previene la falsificación de solicitud entre sitios (CSRF)

Valor devuelto por el métodos de acción

- Se pueden crear métodos de acción que devuelven un objeto de cualquier tipo, como una cadena, un entero o un valor booleano.
- La mayoría de los métodos de acción devuelven una instancia que implemente IActionResult.
- La clase derivada de ActionResult determina el resultado de la acción.
- El controlador hereda métodos auxiliares que simplifican y automatizan la creación de los objetos ActionResult devueltos.
- El resultado más frecuente consiste en llamar al método View. El método View devuelve una instancia de la clase ViewResult, que se deriva de ActionResult.

Tipos derivados de ActionResult

Tipo	Método auxiliar	Descripción
ViewResult	View	Representa una vista como una página web.
PartialViewResult	PartialView	Representa una vista parcial, que define una sección de una vista que se puede representar dentro de otra vista.
RedirectResult	Redirect	Redirecciona a otro método de acción utilizando su URL.
RedirectToRouteResult	RedirectToAction RedirectToRoute	Redirecciona a otro método de acción.
StatusCodeResult	StatusCode NotFound …	Proporciona un modo para devolver un resultado de la acción con un código de estado de respuesta HTTP.

Tipos derivados de ActionResult

Resultado	Método auxiliar	Descripción
ContentResult	Content	Devuelve un tipo de contenido definido por el usuario.
JsonResult	Json	Devuelve un objeto JSON serializado.
FileResult	File	Devuelve la salida binaria para escribir en la respuesta.
EmptyResult		Representa un valor devuelto que se utiliza si el método de acción debe devolver un resultado null (vacío).

Pasar datos a la vista

- El marco de ASP.NET Core MVC proporciona contenedores de nivel de página que pueden pasar datos entre controladores y vistas, débilmente tipados (sin tipo) o fuertemente tipados (modelo).
- La propiedad ViewData es un diccionario (claves/valor) para datos de vista.
- La propiedad ViewBag gestiona el diccionario de datos de vista dinámico (declaración al vuelo).

  ViewBag.p = 1; → ViewData["p"] = 1;
- Vista y controlador comparten las dos propiedades (mismo nombre y al llamar al método View del controlador se asignan a la vista)

Pasar datos fuertemente tipados

- La vista debe heredar de RazorPage<TModel> y establecer le propiedad Model (empezar por):

```
@model Domain.Entity
```

- El modelo se pasa como parámetro al llamar al método View del controlador.
- La propiedad ModelState obtiene el objeto de diccionario de estados del modelo que contiene el estado del modelo y la validación de enlace del modelo.
- El modelo se puede recibir automáticamente:

```
[HttpPost][ValidateAntiForgeryToken]
public ActionResult Edit(Entity ent) {
   if (ModelState.IsValid) {
```

- O manualmente utilizando los métodos del controlador:
  - TryUpdateModel: Actualiza la instancia de modelo especificada con los valores del proveedor de valores actual del controlador.
  - TryValidateModel : Valida la instancia de modelo especificada.

Pasar el estado entre métodos de acción

- Los métodos de acción puede que tengan que pasar datos a otra acción, cuando se produce un error al exponer un formulario, o si el método debe redirigir a métodos adicionales.
- La propiedad TempData es un diccionario (claves/valor) para datos temporales.
- El valor de la propiedad TempData se almacena en el estado de la sesión (proveedor de datos temporales predeterminado) y se conserva hasta que se lea o hasta que se agote el tiempo de espera de la sesión.
- El proveedor de datos temporales se establece en Controller.TempDataProvider.
- Cualquier método de acción al que se llame después de establecer su valor puede obtener valores del objeto y, a continuación, procesarlos o mostrarlos.
- Conservar TempData de esta manera, habilita escenarios como la redirección, porque los valores de TempData están disponibles para más de una única solicitud.

Errores

- Para agregar el mensaje de error especificado a la colección de errores para el diccionario de modelo-estado asociado a la clave especificada.
  - ModelState.AddModelError()
- Para excepciones no tratada se pueden crear filtros de excepciones.
- Para excepciones no tratada se pueden habilitar la página en Startup.Configure:

```
if (env.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
    app.UseDatabaseErrorPage();
} else {
    app.UseExceptionHandler("/Home/Error");
}
```

- Las páginas de excepciones para el desarrollador solo deben habilitarse cuando la aplicación se ejecute en el entorno de desarrollo. Es un riesgo de seguridad compartir públicamente información detallada sobre las excepciones cuando la aplicación se ejecute en producción.

Contexto de ejecución

Propiedad	Descripción
ControllerContext	Obtiene o establece el contexto del controlador.
HttpContext	Obtiene la información específica de HTTP sobre una solicitud HTTP individual.
ModelState	Obtiene el objeto de diccionario de estados del modelo que contiene el estado del modelo y la validación de enlace del modelo.
Request	Obtiene el objeto HttpRequestBase de la solicitud HTTP actual.
Response	Obtiene el objeto HttpResponseBase de la respuesta HTTP actual.
RouteData	Obtiene los datos de ruta de la solicitud actual.
TempData	Obtiene o establece el diccionario para datos temporales.
TempDataProvider	Obtiene el objeto de proveedor de datos temporales que se utiliza para almacenar los datos para la solicitud siguiente.
Url	Obtiene el objeto auxiliar de direcciones URL que se usa para generar las direcciones URL mediante el enrutamiento.
User	Obtiene la información de seguridad del usuario para la solicitud HTTP actual.

Enlazado

- Las capacidades de binding de ASP MVC conforman un potente mecanismo mediante el cual, de manera automática, se obtienen valores para los parámetros de las acciones que se ejecutan buscándolos en la petición que llega del cliente, así como las propiedades del modelo.
- El binding es el proceso que trata de emparejar los valores y parámetros enviados desde el cliente con los parámetros de las acciones que se ejecutan en el servidor.
- Una vez selecciona una acción, se analizan los nombres de los parámetros de la acción y trata de encontrar parejas entre ellos y los campos de formularios, parámetros de ruta, parámetros GET o archivos adjuntos (siguiendo el orden indicado).

Model Binders

- Extensión del framework que permite crear instancias de clases en base a valores enviados por request.
  - Al action llega el objeto instanciado y no los valores del request
- Los controladores Razor y las páginas funcionan con datos procedentes de solicitudes HTTP. La escritura de código para recuperar cada uno de estos valores y convertirlos de cadenas a tipos de .NET sería tediosa y propensa a errores. El enlace de modelos automatiza este proceso. El sistema de enlace de modelos:
  - Recupera datos de diversos orígenes, como datos de ruta, campos de formulario y cadenas de consulta.
  - Proporciona los datos a controladores y páginas Razor en parámetros de método y propiedades públicas.
  - Convierte datos de cadena en tipos de .NET.
  - Actualiza las propiedades de tipos complejos.
- Nos permite participar del ciclo de vida de creación de la instancia permitiéndonos por ejemplo validar los atributos y agregar mensajes de error invalidando el modelo

Enlazado por defecto

- El enlace de modelos intenta encontrar valores para los tipos de destinos siguientes:
  - Parámetros del método de acción de controlador al que se enruta una solicitud.
  - Parámetros del método de controlador Razor pages al que se enruta una solicitud.
  - Propiedades públicas de un controlador o una clase PageModel, si se especifican mediante atributos.
- Por defecto, convenio de nombres:
  - Nombre GET → parámetro del método de acción
  - Nombre POST → parámetro del método de acción
- Tipos complejos:
  - Nombre POST → propiedad del parámetro
- Tipos complejos que contienen propiedades complejas:
  - Nombre POST → propiedad de la propiedad del parámetro
- Tipo lista
  - Nombre POST[n] → parámetro[n]
- Ficheros
  - Nombre POST → parámetro HttpPostedFileBase

Personalizar el enlazado

- El decorador [Bind] se usa para proporcionar detalles cómo debe producir el enlace del modelo a un parámetro.
  - Prefix: Prefijo que se va a usa cuando se represente el marcado para enlazar a un argumento de acción o a una propiedad compleja de modelo.
  - Exclude: Lista delimitada por comas de nombres de propiedades para las que no se permite el enlace.
  - Include: Lista delimitada por comas de nombres de propiedades para las que se permite el enlace (el resto se excluye).
- La creación de Binders personalizados (clase que implemente IModelBinder) proporciona un control total del enlazado. Se decora el parámetro con: [ModelBinder(typeof(MiBinder))]

Personalizar el enlazado

- El atributo [BindProperty] se puede aplicar a una propiedad pública de un controlador o una clase PageModel para hacer que el enlace de modelos tenga esa propiedad como destino:

```
[BindProperty]
public Instructor Instructor { get; set; }
```

- Si el origen predeterminado no es correcto, se puede utilizar uno de los atributos siguientes para especificar el origen:
  - [FromQuery] : obtiene valores de la cadena de consulta.
  - [FromRoute] : obtiene valores de los datos de ruta.
  - [FromForm] : obtiene los valores de los campos de formulario publicados.
  - [FromBody] : obtiene los valores del cuerpo de la solicitud.
  - [FromHeader] : obtiene valores de los encabezados HTTP.
- El atributo [BindRequired], aplicable solo a propiedades del modelo, hace que el enlazado de modelos agregue un error de estado si no se puede realizar el enlace para la propiedad de un modelo. Así mismo [BindNever] impide que el enlazado de modelos establezca la propiedad de un modelo.

Enrutamiento

- Las rutas mapean las URL a los métodos de acción de los controladores y sus parámetros.
- Separadas en fragmentos (/), compuestos por variables ({…}) y/o constantes.
- Ruta por defecto (Startup.Configure):

```
app.UseRouting();
app.UseEndpoints(endpoints => {
   endpoints.MapControllerRoute(
      name: "default",
      pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

- Un parámetro puede ser opcional (aparecer o no):
  - Con valor por defecto: {action=Index}
  - Sin valor: {id?}

Rutas personalizadas

- Soporta tantas rutas como sean necesarias pero deben estar registradas.
- Los nombres de ruta proporcionan un nombre lógico a la ruta. La ruta con nombre se puede usar para la generación de direcciones URL.
- El orden del mapeo importa, las mas especificas primero y las mas generales las últimas (la ruta por defecto la última)
- Toda ruta debe conocer su {controller} y su {action}, en caso de no aparecer en el patrón se deben definir sus valores por defecto.

```
endpoints.MapControllerRoute(name: "blog",
    pattern: "articulos/{**articulo}",
    defaults: new { controller = "Blog", action = "Article" });
```

- MVC enlaza por nombre: nombre variable → nombre parámetro.
- El resto de las variables pueden ser:
  - Obligatorias: sin valor por defecto, deben aparecer en la URL.
  - Opcionales con valor: tienen valor por defecto, si no aparecen en la URL el parámetro toma el valor por defecto.
- Un segmento se puede dividir en varios parámetros: files/{filename}.{ext?}

Rutas personalizadas

- El asterisco * y el asterisco doble ** se puede usar como prefijo de un parámetro de ruta para enlazar con el resto del URI. Debe ser el último parámetro. Con un solo * se sustituyen los / por su escape %2F.
- Las restricciones de ruta se ejecutan cuando se busca la coincidencia y antes de que se conviertan en tokens de valores de ruta. La restricciones se establecen a continuación del nombre separándolas con dos puntos:

```
{id:int:min(1)}
```

- Las restricciones a los parámetros disponibles son:
  - {x:alpha} {x:bool} {x:datetime} {x:decimal} {x:double} {x:float} {x:guid} {x:int} {x:length(6)} {x:length(1,20)} {x:long} {x:max(10)} {x:maxlength(10)} {x:min(10)} {x:minlength(10)} {x:range(10,50)} {x:required} {x:regex(^\d{3}-\d{3}-\d{4}$)}
- Mediante expresiones regulares se pueden establecer restricciones mas complejas a los valores de las variables y se pueden crear restricciones de ruta personalizadas mediante la implementación de la interfaz IRouteConstraint.
- Para restringir el verbo HTTP utilizado se configuran las rutas sustituyendo el método MapControllerRoute por MapGet, MapPost, …

Atributos de Enrutado

- El enrutamiento mediante atributos (anotaciones) utiliza un conjunto de atributos para asignar las plantillas de ruta directamente a controladores y acciones.
- Para activar el sondeo de atributos (si no hay rutas previas):

```
app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
```

- Las rutas de atributo admiten la misma sintaxis en línea que las rutas convencionales para especificar parámetros opcionales, valores predeterminados y restricciones.
- Para establecer la ruta particular para una acción:

```
[Route(“reviews/{reviewId}/edit”)]
public ActionResult Edit(int reviewId) { … }
```

- Las rutas se pueden asociar a verbos HTTP con las especializaciones del atributo Route: HttpGet, HttpPost, HttpPut, HttpDelete, HttpHead, HttpPatch

Atributos de Enrutado

- Para que el enrutamiento mediante atributos sea menos repetitivo, los atributos de ruta del controlador se combinan con los atributos de ruta de las acciones individuales (la ruta en el controlador hace que todas las acciones usen el enrutamiento mediante atributos).

```
[Route("Home")]
public class HomeController : Controller {
   [Route("")][Route("Index")][Route("/")]
   public IActionResult Index() { … }
   [Route("About")]
   public IActionResult About() { … }
```

- Las aplicaciones pueden mezclar el uso de enrutamiento convencional con el de atributos. Es habitual usar las rutas convencionales para controladores que sirven páginas HTML para los exploradores, y mediante atributos para los controladores que sirven las API de REST.

Áreas

- Las áreas son una característica de MVC que se usa para organizar la funcionalidad relacionada en un grupo independiente:
  - Espacio de nombres de enrutamiento para las acciones de controlador.
  - Estructura de carpetas para las vistas.
- El uso de áreas permite que una aplicación tenga varios controladores con el mismo nombre, siempre y cuando tengan áreas diferentes. El uso de áreas crea una jerarquía para el enrutamiento mediante la adición del parámetro de ruta “área”, a controller y action.

```
app.UseEndpoints(endpoints => {
    endpoints.MapControllerRoute(name: "areas",
        pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");
```

- El atributo [Area] es lo que denota que un controlador es parte de un área y, sin el, los controladores no son miembros de ningún área y no coinciden cuando el enrutamiento proporciona el valor de area en la ruta.

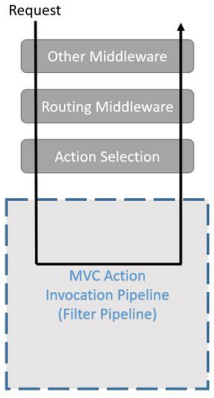
```
[Area("Admin")]
public class HomeController : Controller {
```


Filtrado

- Los controladores definen métodos de acción que normalmente tienen una relación uno a uno con las posibles interacciones del usuario, como hacer clic en un vínculo o enviar un formulario.
- A veces se desea ejecutar la lógica antes de llamar a un método de acción o después de ejecutar un método de acción.
- Los filtros son clases personalizadas que proporcionan un método declarativo y de programación para agregar el comportamiento previo y posterior a la acción a los métodos de acción del controlador.

Filtros

- Los filtros en ASP.NET Core permiten que se ejecute el código antes o después de determinadas fases de la canalización del procesamiento de la solicitud.
- Se pueden crear filtros personalizados que se encarguen de cuestiones transversales, como el control de errores, el almacenamiento en caché, la configuración, la autorización y el registro. Los filtros evitan la duplicación de código. Así, por ejemplo, un filtro de excepción de control de errores puede consolidar el control de errores.
- Los filtros se ejecutan dentro de la canalización de invocación de acciones de ASP.NET Core, a veces denominada canalización de filtro.
- La canalización de filtro se ejecuta después de que ASP.NET Core seleccione la acción que se va a ejecutar.



Tipos de filtros

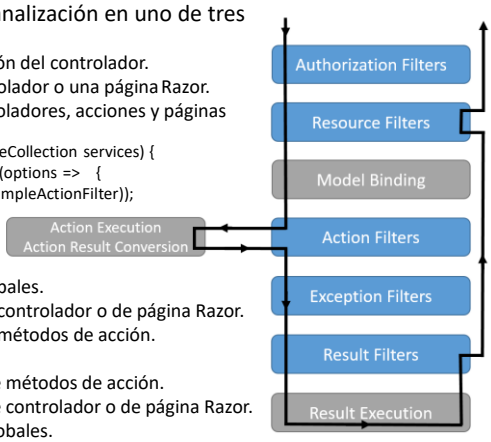
- Filtros de autorización: se ejecutan en primer lugar y sirven para averiguar si el usuario está autorizado para realizar la solicitud, pueden cortocircuitar la canalización si una solicitud no está autorizada.
- Filtros de recursos:
  - Se ejecutan después de la autorización.
  - OnResourceExecuting ejecuta código antes que el resto de la canalización del filtro.
  - OnResourceExecuted ejecuta el código una vez que el resto de la canalización se haya completado.
- Filtros de acción:
  - Ejecutan código inmediatamente antes y después de llamar a un método de acción.
  - Pueden cambiar los argumentos pasados a una acción.
  - Pueden cambiar el resultado devuelto de la acción.
  - No se admiten en páginas Razor.
- Filtros de excepciones: aplican directivas globales a las excepciones no controladas que se producen antes de que se escriba el cuerpo de respuesta.
- Filtros de resultados: ejecutan código inmediatamente antes y después de la ejecución de resultados de acción. Se ejecutan solo cuando el método de acción se ha ejecutado correctamente y son útiles para la lógica que debe regir la ejecución de la vista o el formateador.

Ámbito y Orden de ejecución de filtros

- Un filtro se puede agregar a la canalización en uno de tres ámbitos posibles:
  - Mediante un atributo en una acción del controlador.
  - Mediante un atributo en un controlador o una página Razor.
  - Globalmente para todos los controladores, acciones y páginas Razor:

```
public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews(options =>  {
        options.Filters.Add(typeof(MySampleActionFilter));
    });
}
```

- La secuencia de ejecución:
  1. El código anterior de los filtros globales.
  2. El código anterior de los filtros de controlador o de página Razor.
  3. El código anterior de los filtros de métodos de acción.
  4. Código de la acción.
  5. El código posterior de los filtros de métodos de acción.
  6. El código posterior de los filtros de controlador o de página Razor.
  7. El código posterior de los filtros globales.



Ciclo de ejecución de una acción

- Síncrona:
  - OnActionExecuting: Se le llama antes de invocar al método de acción.
    - Se puede implementar a través de un filtro global, de controlador o de acción.
    - Se puede sobrescribir en el controlador, con ámbito de controlador.
  - OnActionExecuted: Se le llama después de ejecutar el método de acción.
    - Se puede implementar a través de un filtro global, de controlador o de acción.
    - Se puede sobrescribir en el controlador, con ámbito de controlador.

- Asíncrona:
  - OnActionExecuting: Se le llama antes de invocar asíncronamente al método de acción, el callback next se invoca después de ejecutar el método de acción.
    - Se puede implementar a través de un filtro global, de controlador o de acción.
    - Se puede sobrescribir en el controlador, con ámbito de controlador.

Algunos filtros proporcionados en ASP.NET Core MVC

- **Authorize**: Restringe el acceso mediante autenticación y opcionalmente mediante autorización.
- **AllowAnonymous**: Marca controladores y acciones para omitir Authorize durante la autorización.
- **RequireHttps**: Obliga a reenviar las solicitudes HTTP no seguras sobre HTTPS.
- **AcceptVerbs**: Especifica a qué verbos HTTP responderá un método de acción (Con **HttpGet, HttpPost, HttpPut, HttpHead, HttpPatch, HttpOptions** y **HttpDelete** el método solo administra las solicitudes HTTP del verbo).
- **ValidateAntiForgeryToken**: Impide la falsificación de una solicitud.
- **IgnoreAntiforgeryToken**: Ignora la validación contra la falsificación de una solicitud.
- **RequestSizeLimit**: Limita el tamaño máximo del cuerpo de una solicitud:

Construcción vs Uso

- Los sistemas de software deben separar el proceso de inicio, en el que se instancian los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que utilizan las instancias. La separación de conceptos es una de las técnicas de diseño más antiguas e importantes.
- El mecanismo mas potente es la Inyección de Dependencias, la aplicación de la Inversión de Control a la gestión de dependencias. Delega la instanciación en un mecanismo alternativo, que permite la personalización, responsable de devolver instancias plenamente formadas con todas las dependencias establecidas. Permite la creación de instancias bajo demanda de forma transparente al consumidor.

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto a unas técnicas de programación:
  - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
  - en los que la interacción se expresaba de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

Inyección de Dependencias

- Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto, básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase.
- La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementando la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas mediante la suplantación.
- La Inyección de Dependencias proporciona:
  - Código más limpio
  - Desacoplamiento más eficaz, pues los objetos no deben de conocer donde están sus dependencias ni cuales son.
  - Facilidad en las pruebas unitaria e integración

Inyección en ASP.NET Core

- ASP.NET Core admite el patrón de diseño de software de inyección de dependencias (DI), que es una técnica para conseguir la inversión de control (IoC) entre clases y sus dependencias. Una dependencia es un objeto (servicio) del que depende otro objeto.
- El ciclo de vida de la inyección de dependencias es:
  - registrar, resolver y eliminar
- El marco de ASP.NET Core permite administrar el ciclo mediante el registro, resolución y eliminación de dependencia, facilitando el uso de la inyección de dependencia en las aplicaciones.
- Permite dos tipos de inyección
  - inyección del constructor (es la más frecuente)
  - inyección de la llamada a un método
- Para casos excepcionales, permite la resolución manual de dependencias.

Registro

- Los servicios se pueden insertar en el constructor Startup y en el método Startup.Configure.
- Los servicios se pueden registrar con una de las siguientes duraciones:
  - Transitorio (AddTransient): se crean cada vez que el contenedor del servicio los solicita, no se cachea, indicada para servicios sin estado ligeros.
  - Con ámbito (AddScoped): se crean una vez por solicitud del cliente (conexión), indicada para servicios con estado por cliente, se eliminan al final de la solicitud.
  - Singleton (AddSingleton): se crean la primera vez que se solicitan (o al proporcionar una instancia en el registro, rara vez es necesario), instancia única para toda la aplicación y todas las solicitudes de cliente, deben ser seguros para los subprocesos y se suelen usar en servicios sin estado.
- El marco ASP.NET Core usa una convención para registrar un grupo de servicios relacionados: un único método de extensión Add{GROUP_NAME} para registrar todos los servicios requeridos por una característica del marco. Por ejemplo, el método de extensión AddControllers registra los servicios necesarios para los controladores MVC.

Registro

- Aunque se puede registrar un tipo concreto, se recomienda el uso de interfaces o clases base para abstraer la implementación de las dependencias.

```
public void ConfigureServices(IServiceCollection services) {
   // Tipo concreto
   services.AddSingleton<MyClass>();
   // Interfaz
   services.AddSingleton<IMyDependency, MyClass>();{
   // Instancia concreta
   services.AddSingleton<IMyDependency>(new MyClass( … ));
   services.AddSingleton(new MyClass( … ));
   // Duraciones
   services.AddTransient<IOperationTransient, Operation>();
   services.AddScoped<IOperationScoped, Operation>();
   services.AddSingleton<IOperationSingleton, Operation>();
}
```


Resolver

- Los servicios se agregan como un parámetro del constructor de la clase y el marco de trabajo asume la responsabilidad de crear u obtener una instancia de la dependencia y de desecharla cuando ya no es necesaria.

```
public class HomeController : Controller {
   private readonly IMyDependency _myDependency;
   public HomeController(IMyDependency myDependency) {
      _myDependency = myDependency;
   }
```

- Mediante el uso del patrón de DI, el controlador:
  - No usa el tipo concreto, solo la interfaz que implementa. Esto facilita el cambio de la implementación que el controlador utiliza sin modificar el controlador.
  - No crea una instancia, la crea el contenedor de DI.
- No es raro usar la inyección de dependencias de forma encadenada. Una dependencia solicitada puede, a su vez, solicitar sus propias dependencias. El contenedor resuelve las dependencias del grafo y devuelve la instancia totalmente formada.

Resolver

- El atributo [FromServices] permite la inyección de un servicio directamente en un método de acción sin usar la inyección de constructores:

```
public IActionResult About([FromServices] IMyDependency myDependency) {
```

- Para la inyección de servicios desde el main() es necesario crear un elemento IServiceScope con IServiceScopeFactory.CreateScope para resolver un servicio con ámbito dentro del ámbito de la aplicación. Este método resulta útil para tener acceso a un servicio con ámbito durante el inicio para realizar tareas de inicialización.

```
public class Program {
    public static void Main(string[] args) {
        var host = CreateHostBuilder(args).Build();
        using (var serviceScope = host.Services.CreateScope())
            var services = serviceScope.ServiceProvider;
            try {
                var myDependency = services.GetRequiredService<IMyDependency>();
                myDependency.WriteMessage("Call services from main");
            } catch (Exception ex) {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred.");
            }
        }
        host.Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args).ConfigureWebHostDefaults(webBuilder => {
            webBuilder.UseStartup<Startup>();
        });
}
```


REpresentational State Transfer

SERVICIOS RESTFUL

REST (REpresentational State Transfer)

- Un **estilo de arquitectura** para desarrollar aplicaciones web distribuidas que se basa en el uso del protocolo HTTP e Hypermedia.
- Definido en el 2000 por Roy Fielding, para no reinventar la rueda, se basa en aprovechar lo que ya estaba definido en el HTTP pero que no se utilizaba.
- El HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado:
  - HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT
- El HTTP permite en el encabezado transmitir la información de comportamiento:
  - Accept, Content-type, Response (códigos de estado), Authorization, Cache-control, …

Uso de la cabecera

- **Request**: Método /uri?parámetros
  - GET: Recupera el recurso
    - Todos: Sin parámetros
    - Uno: Con parámetros
  - POST: Crea un nuevo recurso
  - PUT: Edita el recurso
  - DELETE: Elimina el recurso
- **Accept**: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
- **Content-type**: Indica en que formato está codificado el cuerpo, utilizando MIME
- **Response**: Código de estado con el que el servidor informa del resultado de la petición.

Peticiones

Request: GET /users
  Response: 200
  content-type:application/json
Request: GET /users/11
  Response: 200
  content-type:application/json
Request: POST /users
  Response: 201 Created
  content-type:application/json
  body
Request: PUT /users/11
  Response: 200
  content-type:application/json
  body
Request: DELETE /users/11
  Response: 204 No Content

Fake Online REST API
for Testing and Prototyping
https://jsonplaceholder.typicode.com/

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de objetos asociados que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
  - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
  - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

SERVICIOS WEB API

Introducción

- ASP.NET Core admite la creación de servicios RESTful, lo que también se conoce como API Web, mediante C#. Para gestionar las solicitudes, una API Web usa controladores.
- Los controladores de una API Web son clases que se derivan de ControllerBase, mediante la cual podemos crear una auténtica capa de servicios REST de manera muy similar a como veníamos desarrollando los controladores hasta ahora.
- Es posible construir una capa de servicios simplemente retornando un JsonResult en las distintas acciones de los controladores.
- La Web API se encarga de realizar este trabajo de forma específica, ofreciendo unas funcionalidades más avanzadas y apropiadas para crear este tipo de capas de servicios de tipo REST.

```
[ApiController]
[Route("api/[controller]")]
public class AlumnosController : ControllerBase
```

Verbos HTTP

- No se basan en los métodos de acción, se basa en los verbos HTTP (8 verbos):
  - GET: Pide una representación del recurso especificado.
  - HEAD: Pide una representación del recurso especificado.
  - POST: Envía los datos en el cuerpo de la petición para que sean procesados.
  - PUT: Sube, carga o realiza un upload a un recurso especificado.
  - DELETE: Borra el recurso especificado.
  - TRACE: Solicita al servidor que envíe de vuelta en el mensaje completado el cuerpo de entidad con la traza de la solicitud.
  - OPTIONS: Devuelve los métodos HTTP que el servidor soporta para un URL específico.
  - CONNECT: Se utiliza para saber si se tiene acceso a un host.

RESTful - CRUD

- Acrónimo de Create, Read, Update, Delete (altas, bajas, modificaciones y consultas)
- Se debe crear un método de acción por cada verbo salvo la consulta que requiere dos métodos: uno sin parámetros para obtener todos y otro con parámetro para obtener solo uno.
- La correspondencia entre verbos HTTP y métodos se puede realizar de dos formas según se usen:
  - Prefijos: El nombre del método comienza con el nombre del verbo que implementa.
  - Decoradores: Hay libertad en la elección del nombre del método pero deben ir precedidos por el decorador del correspondiente verbo.

Verbos y métodos

Acción	Verbo	Prefijo	Decorador	Parámetros	Retorno
Consulta todos	GET	Get	HttpGet	Ninguno	IEnumerable<Entidad> ó HttpResponseException
Consulta uno	GET	Get	HttpGet	Id(Clave)	Entidad ó HttpResponseException
Añadir	POST	Post	HttpPost	Entidad	HttpResponseMessage con HttpStatusCode
Modificar	PUT	Put	HttpPut	Entidad	HttpResponseMessage con HttpStatusCode
Borrar	DELETE	Delete	HttpDelete	Id(Clave)	HttpResponseMessage con HttpStatusCode

Api Controller

```
[Route("api/[controller]")]
[ApiController]
public class RestController : ControllerBase {
   // GET: api/rest
   [HttpGet]
   public IEnumerable<string> Get() { … }
   // GET api/rest/5
   [HttpGet("{id}")]
   public string Get(int id) { … }
   // POST api/rest
   [HttpPost]
   public void Post([FromBody] string value) { … }
   // PUT api/rest/5
   [HttpPut("{id}")]
   public void Put(int id, [FromBody] string value) { … }
   // DELETE api/rest/5
   [HttpDelete("{id}")]
   public void Delete(int id) { … }}
}
```

Métodos de acción

- El filtro ProducesResponseType especifica el tipo del valor y el código de estado devueltos por la acción.

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Persona> Create(Persona item) {
   // …
   return CreatedAtAction(nameof(Get), new { id = item.Id }, item);
}
```

- El controlador, además de StatusCode, cuenta con métodos para generar respuestas especificas: Ok (200), Created(201), CreatedAtAction(201 con cabecera location), Accepted(202), NoContent (204) BadRequest (400), NotFound (404), Conflict (409).
- ASP.NET Core incluye el tipo ActionResult<T> para las acciones del ApiController que permite la conversión implícita de return T; a return new ObjectResult(T); y excluir la propiedad Type de [ProducesResponseType].

Detalles de problemas

- MVC transforma un resultado de error (un resultado con el código de estado 400 o superior) en un resultado con ProblemDetails, basado en la especificación RFC 7807 para proporcionar detalles de error de lectura mecánica en una respuesta HTTP.

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "|7fb5e16a-4c8f23bbfc974667.",
  "errors": {
    "": [  "A non-empty request body is required." ]
  }
}
```

- El método Problem crea un objeto ObjectResult que genera una respuesta ProblemDetails.
- El atributo [ApiController] hace que los errores de validación de un modelo desencadenen automáticamente una respuesta HTTP 400.

Negociación de contenidos

- Los datos del cuerpo de la solicitud pueden estar en XML, JSON u otro formato. Para analizar estos datos, el enlace de modelos usa un formateador de entrada configurado para controlar un tipo de contenido determinado (cabecera content-type) y un formateador de salida para generar un contenido determinado (cabecera accept). Si no dispone del formateador apropiado devuelve 406 Not Acceptable.
- De forma predeterminada, en ASP.NET Core se incluyen formateadores de entrada basados en JSON para el control de los datos JSON. Se pueden agregar otros formateadores para otros tipos de contenido.

```
builder.Services.AddControllers().AddXmlSerializerFormatters();
```

- El filtros [Consumes("application/xml")] permiten que un controlador o una acción limite los tipos de contenido de la solicitud compatibles según su cabecera content-type y [Produces("application/json", "application/xml")] según su cabecera accept.
- Los atributos [Xml..] y [Json…] permiten personalizar la serialización:

```
[XmlAttribute("id")][JsonPropertyName("id")]
public int CustomerId { get; set; }
```


Seguridad

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com
- Las funciones y métodos incluidos en ese código no pueden acceder a:
  - https://www.ejemplo.com/scripts/codigo2.js
  - http://www.ejemplo.com:8080/scripts/codigo2.js
  - http://scripts.ejemplo.com/codigo2.js
  - http://192.168.0.1/scripts/codigo2.js
- La propiedad document.domain se emplea para permitir el acceso entre subdominios del dominio principal de la aplicación.

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece y, por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.
- XMLHttpRequest sigue la política de mismo-origen, por lo que, una aplicación usando XMLHttpRequest solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron a los proveedores de navegadores que permitieran a XMLHttpRequest realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE“
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

Desactivar la seguridad de Chrome

- Pasos para Windows:
  - Localizar el acceso directo al navegador (icono) y crear una copia como “Chrome Desarrollo”.
  - Botón derecho -> Propiedades -> Destino
  - Editar el destino añadiendo el parámetro al final. ej: "C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --disable-web-security
  - Aceptar el cambio y lanzar Chrome
- Para desactivar parcialmente la seguridad:

  --allow-file-access

  --allow-file-access-from-files

  --allow-cross-origin-auth-prompt
- Referencia a otros parametros:
  - http://peter.sh/experiments/chromium-command-line-switches/

Habilitar CORS en ASP.NET Core

- Crear la política en Startup.ConfigureServices (sin nombre sería la predeterminada):

```
services.AddCors(options => { options.AddPolicy("AllowAll", builder =>
        builder.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader()); });
```

- Habilitarla mediante middleware para toda la aplicación en Startup.Configure:

```
app.UseCors("AllowAll");
```

- Habilitarla en el enrutamiento de puntos de conexión:

```
endpoints.MapControllerRoute(name: "apis", pattern:
        "api/{controller:exists}/{id?}", defaults: new { area = "Apis"
        }).RequireCors("AllowAll");
```

- Habilitarla con el atributo [EnableCors] para controladores o métodos de acción concretos.

```
[ApiController]
[EnableCors("AllowAll")]
public class AlumnosController : ControllerBase
```


API mínimas

- Las API mínimas están diseñadas para crear API HTTP con dependencias mínimas. Son ideales para microservicios y aplicaciones que desean incluir solo los archivos, las características y las dependencias mínimas en ASP.NET Core.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.Urls.Add("http://localhost:3000");
app.Urls.Add("http://localhost:4000");

app.MapGet("/", () => "Hello World");
app.MapGet("/oops", () => "Oops! An error happened.");
app.MapGet("/api", (int id) => "This is a GET");
app.MapGet("/api/{id}", () => "This is a GET");
app.MapPost("/api", (DTO item) => "This is a POST");
app.MapPut("/api/{id}", (int id, DTO item) => "This is a PUT");
app.MapDelete("/api", (int id) => "This is a DELETE");

app.MapMethods("/options-or-head", new[] { "OPTIONS", "HEAD" },
            () => "This is an options or head request ");
app.Run();
```