



ASP.NET MVC

Antonia Sacchitella
Analyst @icubedsrl

antonia.sacchitella@icubed.it



Cos'è ASP.NET Core?

Una rivoluzionaria versione di ASP.NET

Basato su .NET Core

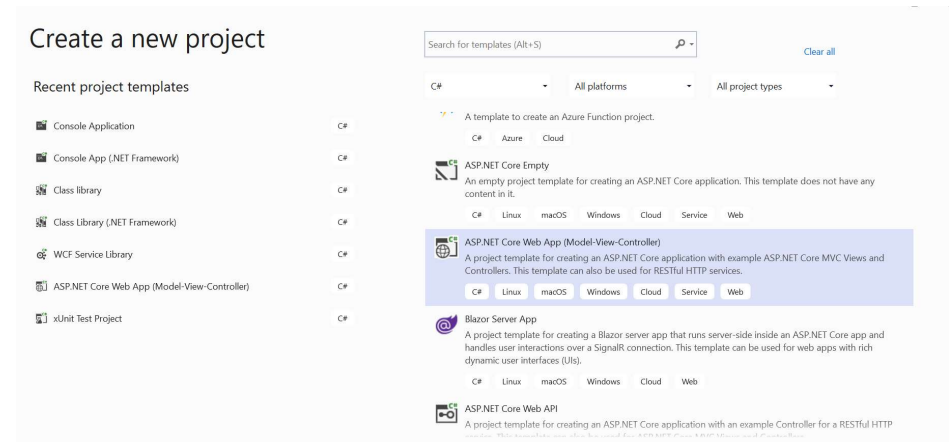
Un taglio netto rispetto al passato

Completamente riscritta





Cross-platform

Born in the cloud

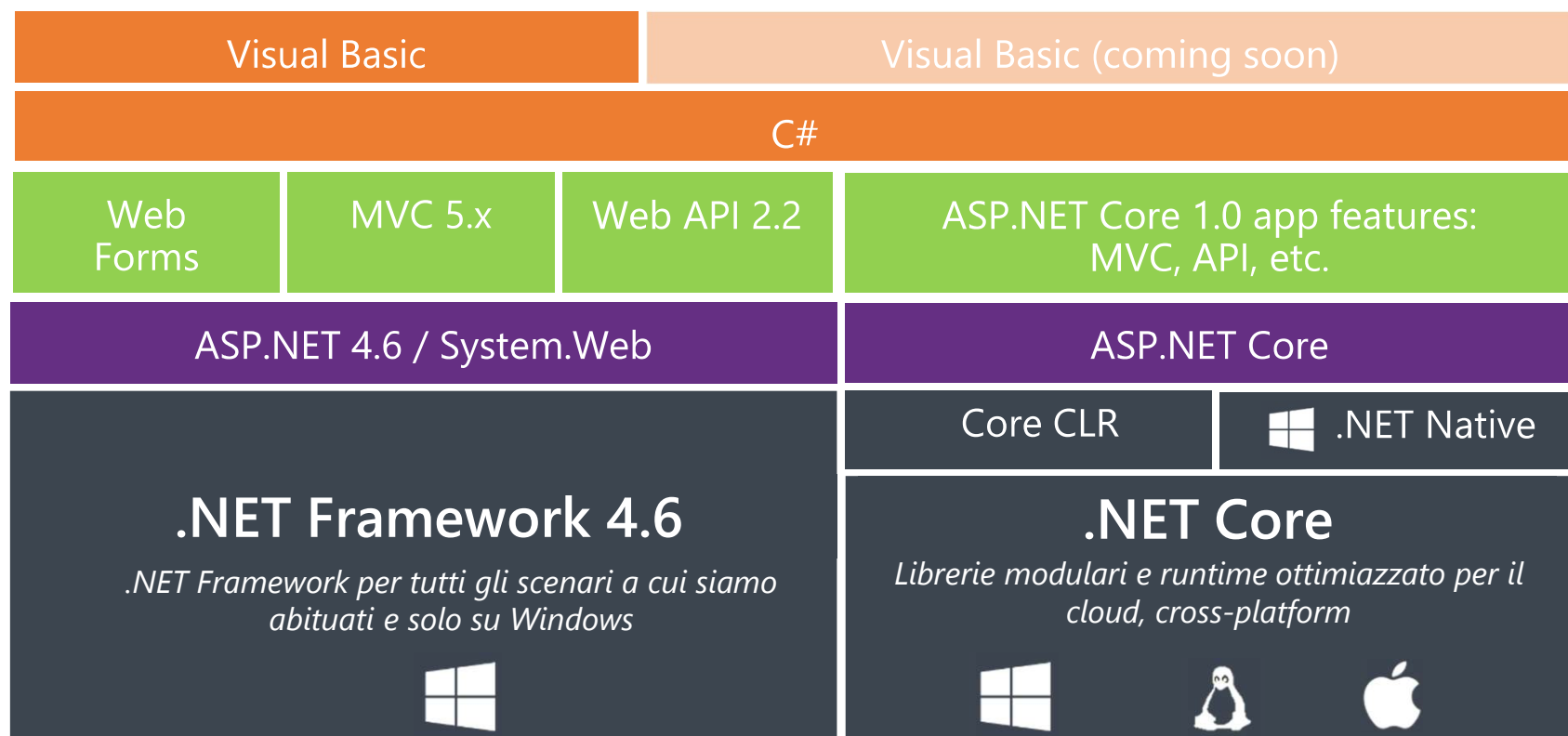
Performante



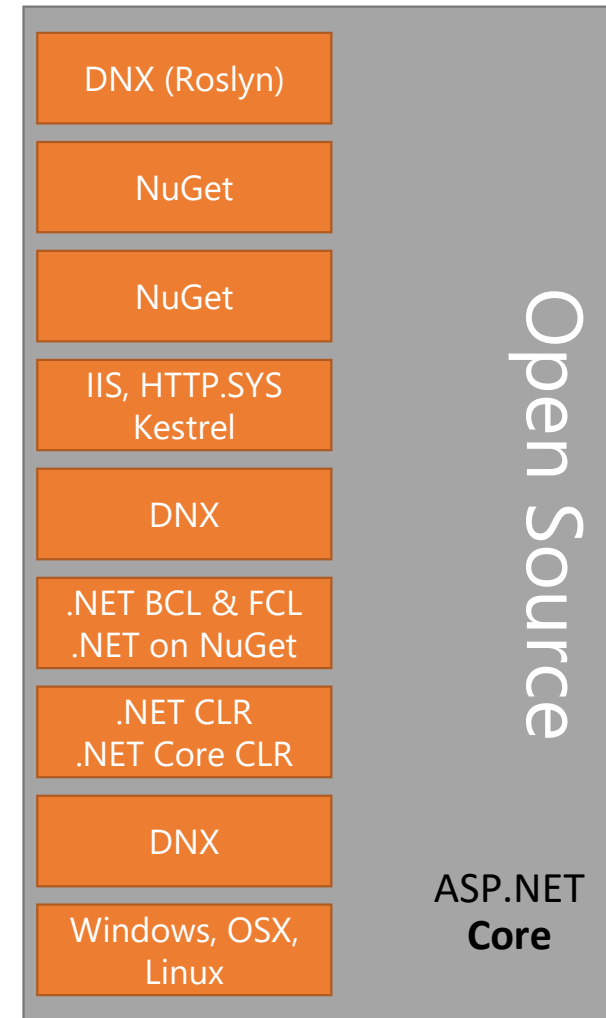
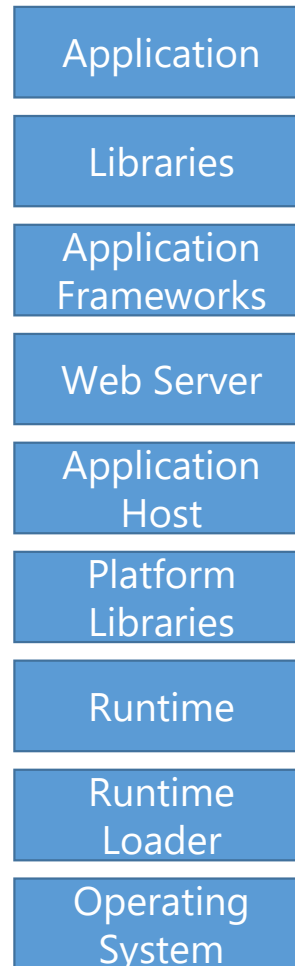
ASP.NET 4.x e ASP.NET Core

ASP.NET 4.x		ASP.NET Core	
.NET Framework 4.x		.NET Core	  
.NET framework libraries		.NET core libraries	
Compilatori e runtime (.NET Compiler Platform: Roslyn, C#, VB, F#, RyuJIT, SIMD)			

ASP.NET



ASP.NET 4.x



ASP.NET Core 101

ASP.NET Core = runtime + ASP.NET MVC + ASP.NET Web API

Usando ASP.NET Core finiamo inevitabilmente ad usare ASP.NET MVC

Addio *web.config*

- Configurazione basata su codice/JSON

Modulare e componentizzabile

Gira su IIS o Self-hosted

- Kestrel

Dependency Injection già integrata

L'app web è di fatto una app console

- Vedi *Startup.cs* dentro il progetto

Server

ASP.NET è disegnato per essere disaccoppiato dal server

Tradizionalmente, ASP.NET è stato legato ad IIS

ASP.NET Core gira su IIS sfruttandolo come reverse proxy

Ci sono due server web:

- IIS Express (default con VS)
- Kestrel

La configurazione è nel file *Program.cs*

IIS vs Kestrel

Usando IIS

- L'app viene eseguita nello stesso processo di lavoro IIS
- Non ha un

Usando Kestrel

- L'app viene eseguita in un processo a parte.
- È l'implementazione predefinita del server HTTP multiplatforma. Kestrel offre le prestazioni migliori e l'utilizzo della memoria ma non dispone di alcune funzionalità avanzate (es. memorizzazione nella cache delle risposte).

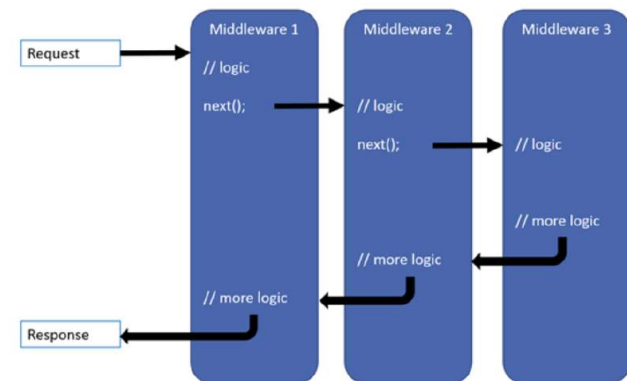
Middleware

Le richieste vengono eseguite una di seguito all'altra

Ogni elemento può interrompere l'esecuzione della pipeline (short-circuiting)

Nel template di default (in sequenza)

- Error handling
- Static file server
- Authentication
- MVC



Startup.cs

Classe che definisce l'entry point

Può stare in un assembly esterno, basta specificare la chiave *Hosting:Application*

Può a propria volta accettare DI nel costruttore

Gestisce la *Configuration* attraverso un *ConfigurationBuilder*

Accende le opzioni di ASP.NET Core

- Es: file statici, MVC, etc

Dependency Injection

- Motore di Inversion Of Control integrato
- Gestione del ciclo di vita (lifetime) dell'istanza
- Possibilità di utilizzare plugin per motori di terze parti
- ASP.NET include già un motore di DI

DI con ASP.NET Core

Per configurare ASP.NET

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddTransient<IMyService, MyService>();
    services.AddScoped<IMyRepository, MyRepository>();
}
```

AddTransient: ogni richiesta di iniezione crea un'istanza. Indicato per servizi stateless.

AddScoped: un'istanza per richiesta HTTP (es: Repository, Context di Entity Framework)

AddSingleton: l'istanza viene creata alla prima necessità di iniezione e poi conservata.

Come usare la DI

Costruttore nei controller

```
public class MyRepository : IMyRepository
{
    private readonly ApplicationDbContext _dbContext;

    public MyRepository(ApplicationDbContext dbContext)
    {
        _dbContext = dbContext;
    }
}
```

Oppure con [FromServices]

```
public IActionResult Index([FromServices] IMyRepository customerRepository)
{
    ...
}
```

Environment

Di default ASP.NET Core ha 3 ambienti preconfigurati (nome case insensitive)

- Development
- Staging
- Production

Il valore attuale viene letto dalla variabile *ASPNETCORE_ENVIRONMENT*

- Definibile nel web.config, nella config, su env variable, Azure, etc

Le informazioni per lo sviluppo sono persistite in un file chiamato *launchSettings.json* dentro la directory *Properties*

HostingEnvironment in Razor

Basta utilizzare il tag *environment* per fare output di pezzi di markup differenti, ad esempio per gestire i CSS diversi tra ambienti

```
<environment names="Development">
    ...
</environment>
<environment names="Staging,Production">
    ...
</environment>
```

Convenzioni per startup

Se esiste una classe con nome *Startup{EnvironmentName}*, viene utilizzata quella al posto di *Startup*

- Es: *StartupDevelopment*

Stesso discorso per il metodo *Configure* che vedremo a breve

Torniamo su Startup.cs

È l'entry point della configurazione

Rappresenta tutto quello che deve essere necessario al runtime per funzionare

Carica servizi, middleware

Configura l'IoC

Gestisce gli environment

Il metodo Configure

Gestisce come ASP.NET risponderà alle richieste

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

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

    app.UseStaticFiles();

    app.UseIdentity();

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

Il metodo ConfigureServices

Per configurare ASP.NET

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

In dettaglio

Use

- Consente di aggiungere funzionalità esterne attraverso il Middleware

ConfigureServices

- Consente di gestire i servizi utilizzati dall'applicazione
- Sfrutta il meccanismo di Dependency Injection di ASP.NET

Gestione degli errori

La gestione degli errori avviene in *Startup.cs/Configure*

Grazie agli Environment, differenziamo il comportamento in base all'ambiente

Negli altri casi, viene utilizzato il Controller specificato, con una view che mostra l'errore

Di default non viene differenziato in base al codice HTTP di errore, ma possiamo utilizzare questo codice per farlo:

```
app.UseStatusCodePagesWithRedirects("~/errors/{0}");
```

ASP.NET Core = ASP.NET (Core) MVC

ASP.NET Core include MVC e WebAPI in un solo motore

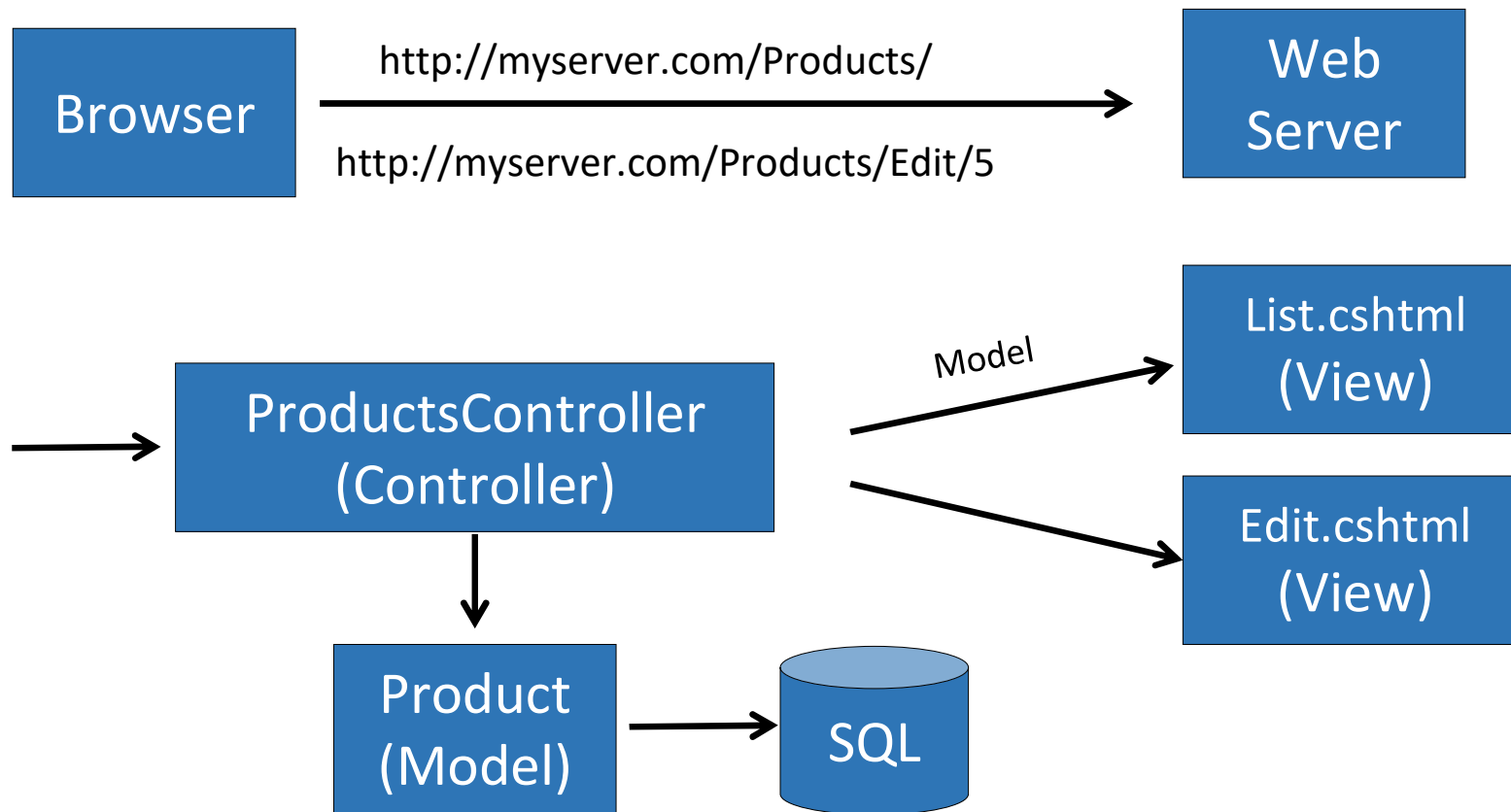
Di fatto, ASP.NET Core, al netto del runtime, è ASP.NET Core MVC

MVC = Model View Controller

- Pattern per disegnare la UI

Rispetto ad ASP.NET MVC, ASP.NET Core MVC ha diverse novità

Come funziona MVC



ASP.NET MVC

MVC non è il 3 tier

- È specifico dell'interfaccia

3 tier è un concetto architetturale

MVC è un concetto di design del software

Model

- Oggetti di business

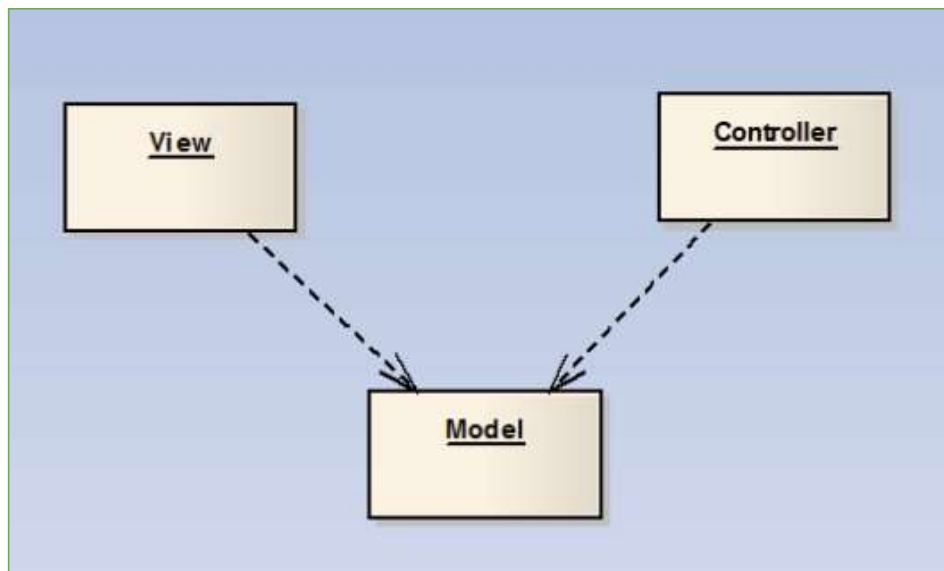
View

- Presentazione

Controller

- Logica

MVC in pillole



Il **Model** ha il compito di contenere i dati da visualizzare ed i metodi che ne permettono l'accesso al nostro engine di persistenza dati

La **View** ha il compito di visualizzare i dati da mostrare nella nostra User Interface

Il **Controller** è il vero cuore, si occupa delle iterazioni con l'utente invocando i metodi presenti nel Model e cambiando l'output della nostra interfaccia tramite la View

Obiettivi di MVC

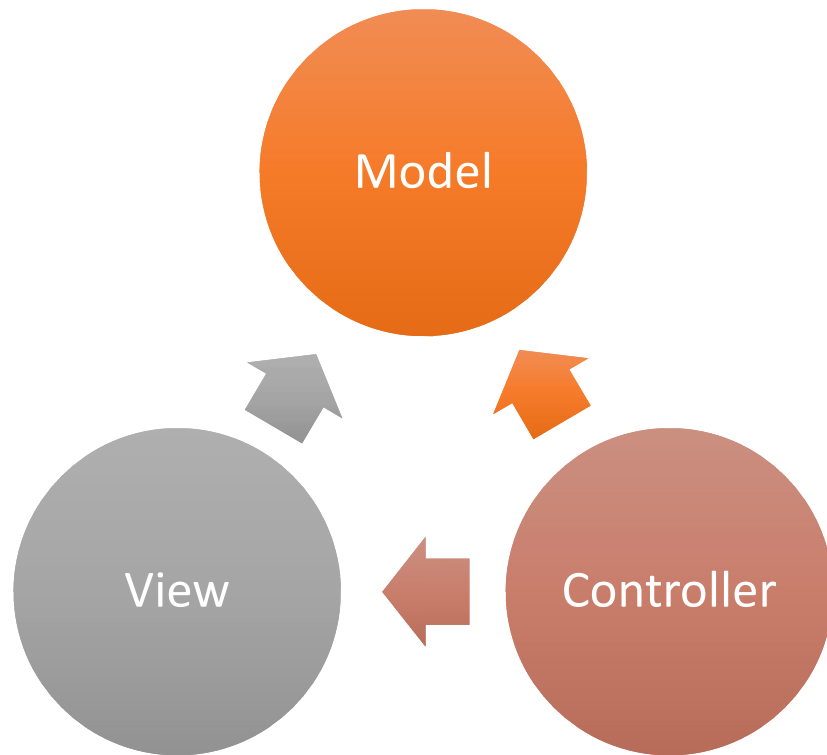
Separation of Concerns

- Testing
- Red/Green TDD
- Alta manutenibilità (tutto è testabile)

Estendibile e pluggabile con maggior facilità

- Con WebForm spesso il modello di estendibilità porta lavoro aggiuntivo (vedi control adapter)

MVC in dettaglio



Il browser richiede /Products/
La Route viene gestita
Il controller viene attivato
Il metodo del controller è invocato
Il controller fa quello che s'ha da fare
Render della View, passando i ViewData

La chiamata di una pagina con MVC

Con ASP.NET normale viene usata la pagina fisica

- A meno che non si usino `HttpHandler`

Con ASP.NET MVC la chiamata viene indirizzata al **Controller**

L'URL Routing interno rigira la chiamata

Il Controller può ereditare da una classe base *Controller* del namespace *Microsoft.AspNetCore.Mvc*.

La View con MVC

Dopo Controller e Model entra in gioco la *View*

La View è una pagina con codice HTML e server side con lo stesso nome dell'action del Controller

- Es List.cshtml, Details.cshtml, etc.

Il dato prelevato è nel Model e passato al Controller tramite il metodo *View*, per visualizzare le informazioni a video

View

Le action solitamente non ritornano semplici stringhe

Normalmente le action restituiscono un'istanza del tipo *ActionResult* o un tipo derivato

Le action sono un componente molto importante dei controller

- Possono restituire una view
- Possono restituire un file
- Possono fare il redirect

La view per convenzione deve stare in

- Views/<ControllerName>/<ViewName>.cshtml
- Views/Shared/<ViewName>.cshtml

Possiamo comunque indicare la view che vogliamo

Novità di ASP.NET Core MVC

ASP.NET Core MVC semplifica il concetto di controller

- Una qualsiasi classe che restituisca in un metodo il tipo *ActionResult* è una *Action*

I controller possono essere POCO

- Plain Old CLR Object, cioè classi che non ereditano per forza da *Controller*

Un controller ha action che restituiscono risposte diverse

- View
- Servizi REST
- Immagini
- Testo
- etc

Demo



Passare dati dal controller alla View

Il Controller può passare dati in modo semplicissimo alla View

Nella Action, è sufficiente istanziare la classe *ViewResult*, passando al costruttore i dati del modello

```
public ActionResult BookView()
{
    return View(new BookViewModel
    {
        Title = "ASP.NET Core 2 Guida completa per lo sviluppatore",
        Author = "Daniele Bochicchio et al",
        Category = "ASP.NET",
        Price = 19.99,
        ReleaseDate = DateTime.Today.AddDays(-40)
    });
}
```

Definizione del Model

Il model è una classe che incapsula i dati ad uso e consumo della View
Non ci sono convenzioni sui nomi

```
public class BookViewModel
{
    public string Title { get; set; }
    public string Category { get; set; }
    public string Author { get; set; }
    public double Price { get; set; }
    public bool OnSale { get; set; }
    public DateTime ReleaseDate { get; set; }
}
```

Recupero del modello nella View

La view può recuperare il modello

```
@model BookViewModel  
  
<h1>@Model.Title</h1>  
<p>Prezzo: @Model.Price.ToString()</p>
```

Se non viene specificato il modello, la View è dinamica e funzionerà comunque (perdiamo l'IntelliSense in Visual Studio)

Passaggio di parametri

Possiamo utilizzare *ViewData* e *ViewBag*

```
ViewData["Message"] = "messaggio";  
ViewBag.Message = "messaggio";
```

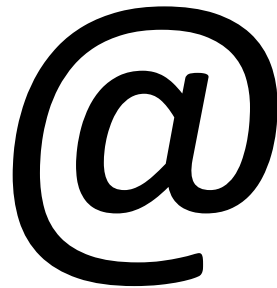
ViewBag prevede un accesso dinamico al contenuto del *ViewData*, quindi non richiede casting

Sostanzialmente sono la stessa cosa

View e Razor

Razor è il View engine di default per ASP.NET Core MVC

La sua sintassi è basata su C#



Istruzione su singola riga

```
@{ var theMonth = DateTime.Now.Month; }  
<p>Valore numerico del mese corrente: @theMonth</p>
```

Istruzione multi riga

```
@{  
    var outsideTemp = 35;  
    var weatherMessage = "Ciao, ci sono " + outsideTemp + " gradi.";  
}  
  
<p>Previsioni di oggi: @weatherMessage</p>
```

Ogni riga di codice, termina come in C# con il ;

È possibile dichiarare le variabili, utilizzando **var**

Uso di \ in una stringa

```
@{ var myFilePath = @"C:\MyFolder\"; }  
<p>Path: @myFilePath</p>
```

Le stringhe devono essere racchiuse tra virgolette

In caso la stringa rappresenti un percorso dobbiamo utilizzare l'operatore @, esattamente come in C#

Virgolette nelle stringhe

```
@{ var myQuote = @"Il mio libro preferito è: ""ASP.NET Core""; }  
<p>@myQuote</p>
```

Se le stringhe contengono le virgolette è sufficiente raddoppiarle

Sintassi e tip

```
@if (IsPost) {  
    <p>Ciao, oggi è @DateTime.Now e questo è un post!</p>  
}  
else {  
    <p>Ciao <em>straniero</em>, oggi è: <br /> </p> @DateTime.Now  
}
```

Razor permette di innestare codice client con codice server

Sintassi e tip

```
@if (IsPost) {  
    @: Oggi è: <br /> @DateTime.Now  
    <br />  
    @DateTime.Now @:is the <em>current</em> time.  
}
```

È possibile inserire testo semplice in linea utilizzando @

È possibile anche inserire più righe, semplicemente precedendo ogni riga con @

Sintassi e tip

```
@* Commento singolo. *@  
  
@* Commento multi riga.  
   Generalmente continuerà su altre righe ancora. *@  
  
@{  
    @* Questo è un commento in linea. *@  
    var theVar = 17;  
}
```

È possibile aggiungere commenti sia nell'HTML che nel codice

In alternativa in un blocco di codice è possibile utilizzare la sintassi di C#

Da VS -> CTRL+K, CTRL+C – CTRL+K, CTRL+U

Operatori e loop

if/else/else if

switch

for

foreach

while

Necessitano sempre di un blocco di { } in cui racchiudere il codice

Demo



Tag Helper: che cosa sono

I tag helper permettono al codice server-side di generare codice HTML

Possono essere invocati in base al nome dell'elemento o dell'attributo

I tag helper sono come gli html helper, ma riducono l'utilizzo di sintassi C# all'interno della pagina

- Codice più leggibile e mantenibile

- Supporto all'IntelliSense di Visual Studio

```
<label asp-for="Email"></label>
```

```
<label for="Email">Email</label>
```

Partial View

View parziale (può essere tipizzata) e condivisa

Le view parziali devono essere salvate dentro la directory Shared o in quella delle View

- Sarà una partial view globale o solo di un gruppo di view

Consente di evitare la duplicazione di codice

Per convenzione si preferisce utilizzare il carattere underscore come prefisso di tutte le partial view

```
@Html.Partial("_Partial", Model)

<partial name="_Partial" model="Model" />
```


PartialView async

ASP.NET Core introduce il supporto per le partial async

Utile per migliorare la scalabilità, qualora ci sia codice async nella view

```
@await Html.PartialAsync("_List")
```

Sfruttare le layout view nel progetto

Consente di centralizzare un layout comune a tutte le view

Viene chiamata (per convenzione) è *_Layout.cshtml*

Non è una view tipizzata, perché sarà utilizzata in un gran numero di situazioni e vogliamo lasciare alle singole action la flessibilità di utilizzare il model più consono alla view che dovrà essere mostrata

Sfruttare le layout view nel progetto

```
<!DOCTYPE html>
<html>
<head>... </head>
<body>
  <div id="body">
    @RenderBody()
  </div>
</body>
</html>
```

Nella pagina di layout `@RenderBody` determina dove il contenuto deve essere renderizzato

Sfruttare le layout view nel progetto

Nella pagina inseriamo il contenuto e specifichiamo la pagina di layout

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@model MyBookShop.Models.BookViewModel
@{ ViewBag.Title = Model.Title; }

<h2>Libro</h2>
<p>
    @Model.Title
</p>
<p>
    @Model.Author
</p>
<p>
    @Model.Category
</p>
```

Sfruttare le layout view nel progetto

Impostare ogni volta la pagina di layout può essere tedioso e ripetitivo

Limita la manutenzione

All'interno di *Views_ViewStart.cshtml* imposta globalmente la Layout View per tutte le pagine

Basta sovrascriverla localmente nella View per non utilizzarla

Definire sezioni aggiuntive in una layout view

Possiamo definire nuove sezioni

- anche opzionali

_Layout.cshtml

```
<footer>
  @RenderSection("Footer")
  @RenderSection("OptionalSection", true)
</footer>
```

View.cshtml

```
@section Footer {
  <p>Footer della content view</p>
}
```

Verifica dell'esistenza di una sezione

```
<div id="footer">
  @if (IsSectionDefined("Footer"))
  {
    @RenderSection("Footer")
  }
  else
  {
    <p>Contenuto di default definito su layout view</p>
  }

  @RenderSection("OptionalSection", required: false)
</div>
```

È possibile verificare l'esistenza della sezione e, in alternativa, mostrare del contenuto standard

Importazione globale per View

Possiamo registrare funzionalità a livello globale nel file *Shared_ViewImports.cshtml*

```
@using WebApplication1
@using WebApplication1.Model
@using Microsoft.AspNetCore.Identity
@using Microsoft.AspNetCore.Mvc.Razor
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```


Il file `_ViewStart`

Possiamo eseguire codice per ogni View aggiungendolo all'interno del *Shared_ViewStart.cshtml*

Generalmente viene sfruttato per registrare globalmente una layout page

Environment

Possiamo diversificare il markup in base all'ambiente

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment exclude="Development">
  <link rel="stylesheet"
    href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
    asp-fallback-test-class="sr-only"
    asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

Gestione degli ambienti

ASP.NET Core utilizza una particolare variabile d'ambiente che serve a descrivere l'ambiente in uso.

ASPNETCORE_ENVIRONMENT

Development, Staging, Production, Custom

Gestione degli ambienti

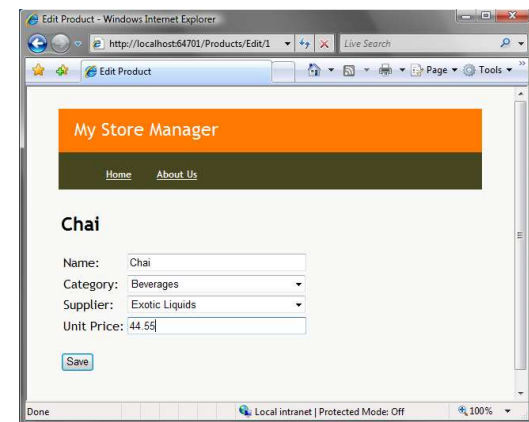
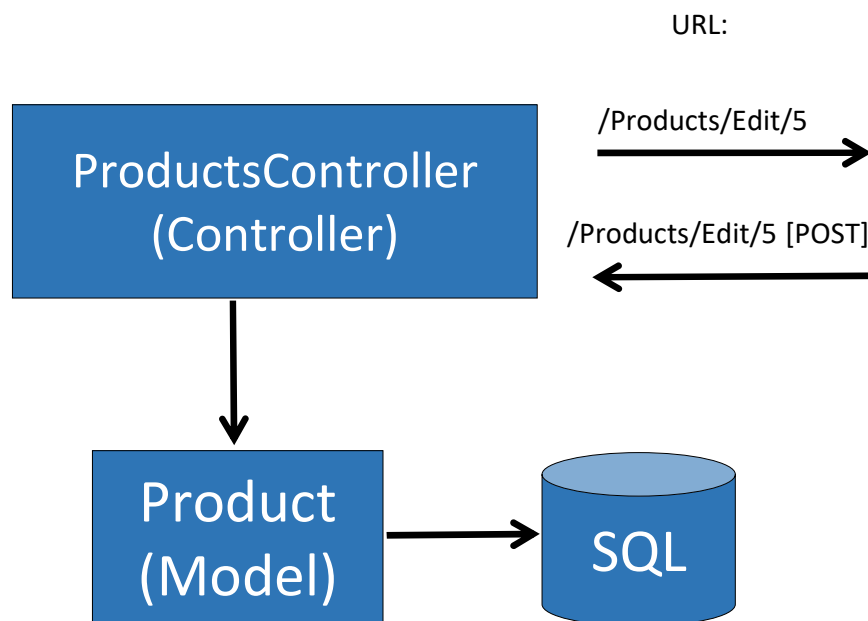
L'accesso via HTML agli ambienti è consentito attraverso l'uso del tag helper di environment

```
<environment names="Development">
  <link rel="stylesheet" href="/dev/bootstrap.css" />
  <link rel="stylesheet" href="/dev/site.css" />
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet" href="/prod/bootstrap.css" />
  <link rel="stylesheet" href="/prod/site.css" />
</environment>
```

Demo



Come funziona l'interazione: le form



Gestire le form

Eliminare l'utilizzo degli HTML helper semplifica il codice generato, soprattutto all'interno delle form di inserimento dati

Grazie agli helper per controller ed action, si può semplificare la navigazione durante il metodo POST

```
<form asp-controller="Demo"  
      asp-action="Register"  
      asp-route="CustomRoute"  
      method="post">  
  <!-- Input and Submit elements -->  
</form>
```

Attributo di input

Imposta l'attributo type in base alla proprietà nel modello e alla DataAnnotation associata (tranne se è esplicitamente impostato)

Genera la validazione secondo gli standard HTML5 in base al modello con gli attributi *data-val-**

E' l'equivalente di *Html.EditorFor/TextBoxFor*

```
<input asp-for="<Expression Name>" />
```


Attributo di input

Corrisponde a

```
<form method="post" action="/">
  <select id="Country" name="Country">
    <option value="MX">Mexico</option>
    <option selected="selected" value="CA">Canada</option>
    <option value="US">USA</option>
  </select>
  <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

La validazione: validation message

Il tag helper di input aggiunge gli attributi di validazione client-side in base alle DataAnnotation trovate sul modello.

La validazione avviene anche lato server nel caso che il javascript sia disattivato nel browser

```
<span asp-validation-for="Email"></span>
```

Validazione del modello

- `DataType` (Specifica il tipo di una proprietà)
- `DisplayName` (Specifica il nome con cui si vuole visualizzare i dati nella view)
- `DisplayFormat` (Specifica un particolare formato di visualizzazione dei dati
Es. Email, Password, Date ecc.)
- `Required` (Campo richiesto)
- `StringLength` (Specifica lunghezza minima e massima di una stringa)

La validazione: validation message

```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email Address")]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}
```

La validazione: validation message

```
@model RegisterViewModel
```

```
<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
```

```
  <div asp-validation-summary="ModelOnly"></div>
```

```
  Email: <input asp-for="Email" /> <br />
```

```
  <span asp-validation-for="Email"></span><br />
```

```
  Password: <input asp-for="Password" /><br />
```

```
  <span asp-validation-for="Password"></span><br />
```

```
  <button type="submit">Register</button>
```

```
</form>
```

La validazione: validation summary

Viene utilizzato per mostrare un report dei messaggi di validazione
E' il corrispondente di *Html.ValidationSummary*

```
<div asp-validation-summary="ModelOnly"></div>
```

Tag helper: select

Genera un tag `<select>` con le opzioni specificate all'interno della proprietà `Items`

E' l'alternativa di *Html.DropDownListFor*

```
<select asp-for="Country"  
        asp-items="Model.Countries"></select>
```

Demo



Esercitazione n. 1

- Realizzare un'applicazione Web sfruttando il costrutto MVC per la gestione di un e-commerce.
- Realizzare un modello dati che comprende un'entità **Prodotto**:
 - *Codice (string), Descrizione (string), Tipologia (del tipo 'Elettronica', 'Abbigliamento', 'Casalinghi'), Prezzo al pubblico (decimal), Prezzo dal Fornitore (decimal).*
- Realizzare un layer dati con Entity Framework per la gestione delle operazioni CRUD sull'entità **Prodotto** (creare un nuovo database *Amazon*)
- Attraverso l'utilizzo di pagine Razor visualizzare:
 - Una Home Page
 - Una pagina dedicata alla visualizzazione della lista dei prodotti (evitando di esporre il prezzo al fornitore)
 - Su ogni riga contenente l'informazione del prodotto aggiungere un pulsante che porta ai dettagli del prodotto stesso.
 - Form per la creazione e l'edit dei prodotti
- Utilizzare due fogli di stile differenti a seconda che l'applicativo venga eseguito in ambiente «Development» o meno.