**30 Controller Riepilogo della sezione**

Realizzare applicazioni web per le persone

La maggior parte delle applicazioni web è composta da varie **pagine HTML dal contenuto dinamico** che gli utenti sfogliano dal loro browser preferito. Anche quando dobbiamo realizzare un'applicazione web che si compone di molte pagine web, **grazie ad ASP.NET Core possiamo mantenere il progetto ben strutturato**.

Esistono due tecnologie ugualmente valide per realizzare questo tipo di applicazioni, che possiamo anche chiamare "siti web":

* ASP.NET Core MVC
* ASP.NET Core Razor Pages

Entrambi sono molto simili e **si basano sulle stesse fondamenta tecnologiche**. Le loro differenze risiedono più che altro nel modo in cui sono organizzati i file nel progetto. Nessuna di esse è superiore all'altra in senso stretto per cui, come sviluppatori, potremo **scegliere liberamente l'una o l'altra tecnologia** in base alle nostre preferenze. Possiamo anche usarle entrambe nello stesso progetto. Forte!  
  
A prescindere dalla nostra preferenza personale, **vale la pena di conoscere approfonditamente entrambe le tecnologie**.

Il  pattern MVC

In questa sezione abbiamo cominciato a esplorare ASP.NET Core MVC e molti dei concetti che affronteremo si applicheranno anche a Razor Pages, che vedremo più avanti.

ASP.NET Core MVC e Razor Pages promuovono la **separazione delle responsabilità**:

* Il **Model** è composto sia di servizi regolano l'accesso ai dati che di classi per lo scambio dei dati stessi;
* Le **View** permettono di presentare i dati in HTML, in modo che possano essere visualizzi nel browser;
* I **Controller** fanno da coordinatori: ricevono i dati dagli utenti e sfruttano le classi del Model per ottenere i dati richiesti. Infine, possono avvalersi delle View per presentarli in HTML.

Model, View e Controller li troviamo all'interno del progetto in directory che recano lo stesso nome (/Models, /Views, /Controllers). Quindi, i nostri file si trovano in directory separate in base alla loro responsabilità. Con ASP.NET Core Razor Pages, invece, si preferisce tenere le "pagine" (che svolgono ruoli equivalenti a quelli della View e del Controller) all'interno della stessa directory /Pages.

*Nota: Le classi del Model possono anche trovarsi in un progetto separato. Infatti, dato che tali classi spesso contengono logica di business che deve restare immutata indipendentemente dal progetto in cui viene usata, potremmo anche creare un progetto Class Library che poi referenziamo come pacchetto NuGet sia dall'applicazione ASP.NET Core che da un'eventuale app mobile Xamarin o altra applicazione per .NET Core.*

Aggiungere ASP.NET Core MVC al progetto

ASP.NET Core MVC è una tecnologia di utilizzo molto comune, perciò Microsoft ha deciso di creare un apposito template chiamato mvc. Digitando il seguente comando, potremo creare una nuova applicazione con ASP.NET Core MVC già integrato.

dotnet new mvc

Comunque, dato che noi siamo partiti dal template web che rappresenta un'applicazione minimale, possiamo aggiungere ASP.NET Core MVC anche dopo, come si vede qui di seguito.

Dal metodo Configure della classe Startup, **aggiungiamo il middleware di routing come ultimo**. Il vecchio middleware che produceva il testo "Hello *nome*" va rimosso.

1. app.UseMvc(routeBuilder => {
2. routeBuilder.MapRoute("Default", "{controller=Home}/{action=Index}/{id?}");
3. });

Come si vede, abbiamo anche indicato una *route*, cioè un template come {controller=Home}/{action=Index}/{id?} che **aiuterà il middleware di routing ad esaminare il percorso della richiesta** e a capire quale controller, quale action e quale parametro *id* saranno usati per rispondere alla richiesta dell'utente. Nel template sono indicati anche dei valori di default come *Home* e *Index* che vengono usati per sopperire alla mancanza dei rispettivi valori nel percorso della richiesta. Ad esempio, visitando il percorso / da browser, è come se avessimo visitato /Home/Index. Il simbolo ? rende un frammento opzionale.

Poi, dal metodo ConfigureServices della classe Startup, aggiungiamo i servizi di ASP.NET Core MVC che il middleware di routing userà per **trovare e costruire il Controller e poi per invocare l'action richiesta**.

1. services.AddMvc();

Ora ASP.NET Core MVC è stato integrato nell'applicazione e siamo pronti a scrivere il primo controller.

Il controller e le action

Un controller è una semplice classe C# che deriva dalla classe base Controller e che creiamo all'interno della directory /Controllers.  
Per **convenzione**, diamo alla classe un nome con il suffisso *Controller*, ad esempio CoursesController.

I metodi pubblici che creiamo all'interno del controller si chiamano **action**. In questo esempio, il CoursesController ha due action chiamate Index e Detail.

1. public class CoursesController : Controller
2. {
3. public IActionResult Index()
4. {
5. return Content("Sono Index");
6. }
7. public IActionResult Detail(string id)
8. {
9. return Content($"Sono Detail e l'id che ho ricevuto è {id}");
10. }
11. }

Grazie alla presenza del middleware di routing e dei servizi di ASP.NET Core MVC, queste due action verranno eseguite quando inviamo delle richieste a /Courses/Index e /Courses/Detail/5.

Solo il controller è strettamente necessario a produrre una risposta. Ben presto però avremo bisogno anche del model e delle view se vogliamo che la nostra applicazione web produca dei contenuti dinamici significativi, anziché delle semplici stringhe di testo statico come in questo esempio.

L'action di un controller può restituire **vari tipi di risultato**:

* Con return Content(*testo*) produciamo una risposta con del testo semplice, come nell'esempio precedente;
* Con return View(*oggetto*) coinvolgiamo una view affinché presenti in HTML un oggetto chiamato viewModel;
* Con return File(percorsoFile) restituiamo il contenuto binario di un file presente su disco;
* Con return Redirect(*url*) e varie sue incarnazioni, eseguiamo un redirect a un altro indirizzo, ad esempio per riportare un docente alla pagina di elenco dei corsi dopo aver salvato un nuovo corso;
* Con return Json(*oggetto*) restituiamo del contenuto JSON che useremo lato JavaScript per supportare l'esperienza lato client, ad esempio per costruire una casella di ricerca con suggerimenti mentre si digita.

Impostare la Compatibility Version

Ogni volta che Microsoft aggiorna ASP.NET Core (ad esempio passando dalla versione 2.1 alla 2.2) **introduce delle modifiche ai meccanismi di funzionamento interno** dei servizi di ASP.NET Core MVC che ne migliorano le prestazioni o ottimizzano il consumo delle risorse. In rari casi, soprattutto se nella nostra applicazione siamo andati a interferire con i componenti interni di ASP.NET Core (cosa che *molto* raramente capita di fare), **queste migliorie possono causare dei lievi effetti collaterali**.

Per capire se la nostra applicazione è interessata, possiamo consultare il [blog ufficiale di ASP.NET](https://blogs.msdn.microsoft.com/webdev/).

Con ASP.NET Core 2.2, ad esempio, si legge in [questa pagina del blog](https://blogs.msdn.microsoft.com/webdev/2018/08/27/asp-net-core-2-2-0-preview1-endpoint-routing/) che il funzionamento interno del middleware di routing è stato completamente rivisto per essere più performante. Se non abbiamo interferito a basso livello con il middleware di routing, possiamo cominciare a sfruttare le nuove migliore impostando la Compatibility Version dal metodo Configure della classe Startup. Andiamo a integrare così la registrazione dei servizi di ASP.NET Core che avevamo fatto in precedenza.

1. services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

Microsoft, infatti, ci chiede di essere espliciti per evitare che vengano introdotti effetti collaterali quando aggiorniamo la versione di ASP.NET Core che usiamo nella nostra applicazione.