**Riepilogo della sezione**

Il model

Nel pattern MVC, la "M" è rappresentata dal "Model" che contiene la **logica per regolare l'accesso ai dati** e, in generale, al mondo esterno all'applicazione. Le classi contenute nel model sono usate, per esempio, dai controller che le sfruttano per ottenere i dati che dovranno essere presentati dalle view.

**Non esiste un unico modo per organizzare la logica** all'interno del model. L'approccio più appropriato dipende molto dal tipo di progetto che si va a realizzare, dalle competenze del team di sviluppo dalla sua volontà a sposare tale approccio.

Organizzazione della logica nel model

In questo corso, la logica all'interno del model è organizzata su classi che hanno scopi diversi:

* I **servizi applicativi** sono le classi con cui i controller interagiscono. Queste classi contengono dei metodi come GetCourses() che serve a recuperare l'elenco dei corsi, oppure GetCourseById(string id) per recuperare le informazioni dettagliate. Nel nostro progetto, il servizio applicativo è consapevole che nell'applicazione tali dati saranno visualizzati in una view e perciò restituisce degli oggetti chiamati *ViewModel* che contengono tutte e sole le proprietà necessarie alla view;
* I **servizi infrastrutturali** sono classi che lavorano a un livello più basso e si occupano di dialogare con il mondo esterno, come stabilire una connessione al database o spedire e-mail, così come disposto dal servizio applicativo che li usa;
* I **ViewModel** sono classi prive di logica e contengono semplicemente delle proprietà. Sono usate per trasportare dati dal servizio applicativo al controller, che poi girerà questi oggetti alla view in modo che siano presentati in HTML;
* **Un insieme di varie altre classi** che creeremo strada facendo, come *value objects* e *entità*.

A prescindere da quale approccio si scelga, è consigliabile che la logica all'interno del model sia organizzata in **piccole classi dalla responsabilità specifica** perché in questo modo saranno facilmente riutilizzabili. Inoltre, l'applicazione resterà modulare e manutenibile nel tempo, anche quando crescerà in complessità.

Usare un servizio applicativo dal controller

Un controller può semplicemente creare un'istanza di un servizio applicativo come CourseService e usare i suoi metodi in questo modo.

1. public class CoursesController : Controller
2. {
3. public IActionResult Index()
4. {
5. var courseService = new CourseService();
6. List<CourseViewModel> courses = courseService.GetCourses();
7. return View(courses);
8. }
9. }

Notiamo come la logica situata all'interno dell'action del controller sia veramente poca: l'action si limita a invocare il metodo GetCourses() del servizio applicativo CourseService per ottenere una lista di CourseViewModel, che passa immediatamente alla view in modo che la lista sia presentata in HTML.  
  
Nella prossima sezione scopriremo un modo più raffinato di ottenere un'istanza del CourseService, sfruttando la *dependency injection* di ASP.NET Core.

View fortemente tipizzate

Ogni view può dichiarare qual è il tipo di dato che può presentare. Per questo, aggiungiamo la direttiva @model (con la m minuscola) in cima al file .cshtml.

1. @model List<CourseViewModel>

In questo modo avremo delle agevolazioni durante lo sviluppo:

* Nella view, usiamo l'espressione @Model (con la M maiuscola) per accedere alle proprietà dell'oggetto passato dal controller che, in questo caso, è un oggetto di tipo List<CourseViewModel>. Ad esempio potremmo stampare il numero di elementi nella lista in questo modo:
  1. <p>Ci sono @Model.Count corsi</p>
* Quando digitiamo @Model. apparirà l'intellisense, cioè i suggerimenti che ci permettono di vedere quali proprietà esistono all'interno dell'oggetto che è stato passato dal controller. In questo modo riduciamo drasticamente il numero di errori di digitazione commessi;
* Se, a causa di un bug, il controller dovesse fornire un oggetto di tipo diverso da quello dichiarato dalla view con la direttiva @model, ce ne accorgeremmo subito, già in fase di compilazione.

ViewData e ViewBag per passare informazioni aggiuntive

Il controller può fornire delle informazioni aggiuntive alla view usando gli oggetti ViewData e ViewBag. Tuttavia, si preferisce non usarli in questo modo perché il compito è già assolto dal ViewModel, che è una classe che abbiamo creato per una view specifica e che perciò dovrebbe contenere tutto quello di cui quella view ha bisogno.

Piuttosto, gli oggetti ViewData e ViewBag dovrebbero essere usati per scambiare informazioni tra la view di contenuto e la view di layout. Un caso tipico è quello del **titolo di pagina**: ogni view di contenuto ha il proprio titolo che deve anche essere visualizzato all'interno del tag <title> che si trova nella view di layout.  
  
Gli oggetti ViewData e ViewBag**hanno esattamente lo stesso scopo** e ciò che cambia tra i due è solo la sintassi da usare. Possiamo usare l'uno o l'altro, a nostro piacimento. Andiamo dunque a usare il ViewData in una **view di contenuto** per impostare il titolo.

1. @{
2. ViewData["Title"] = "Catalogo corsi";
3. }
4. <h1>@ViewData["Title"]</h1>
5. <p>Contenuto della pagina di elenco dei corsi</p>

Se volessimo usare il ViewBag, la sintassi cambierebbe leggermente ma l'esito sarebbe identico.

1. @{
2. ViewBag.Title = "Catalogo corsi";
3. }
4. <h1>@ViewBag.Title</h1>
5. <p>Contenuto della pagina di elenco dei corsi</p>

Ora andiamo nella **view di layout** e mostriamo il titolo nel tag <title> ottenendolo dal ViewData.

1. <html>
2. <head>
3. <title>@ViewData["Title"]</title>
4. </head>
5. <body><!-- Contenuto del body --></body>
6. </html>

Oppure ottenendolo dal ViewBag:

1. <html>
2. <head>
3. <title>@ViewBag.Title</title>
4. </head>
5. <body><!-- Contenuto del body --></body>
6. </html>

Così riusciamo a fare in modo che la view di contenuto e la view di layout possano scambiarsi informazioni, anche se risiedono su file diversi.