Le view Razor

Nel pattern MVC, le view **hanno la responsabilità di presentare in HTML i dati** che vengono forniti dal controller.

Oltre al codice HTML, le view possono contenere codice C# a supporto della presentazione dei dati (cicli for per scorrere una lista, blocchi if per mostrare/nascondere una porzione di markup, e così via). Ecco un esempio.

1. <section>
2. <h1>Elenco dei corsi</h1>
3. <ul>
4. @for(int i = 1; i <= 3; i++)
5. {
6. <li>Corso @i</li>
7. }
8. </ul>
9. </section>

Grazie al simbolo @ possiamo eseguire istruzioni C# (come nel caso dei ciclo for) o stampare il valore di variabili (come nel caso di i), in modo che possiamo generare contenuto HTML in maniera dinamica.  
  
In ASP.NET Core MVC, le view sono dei file che hanno un'estensione .cshtml che indica per l'appunto che in esse possiamo inserire sia codice HTML che codice C#.  
  
**Le view devono essere "stupide"**. Per nessun motivo dovrebbero contenere codice di accesso al database, di invio e-mail, o di qualsiasi altra attività che richiede la comunicazione con il mondo esterno perché **queste sono responsabilità del model**. Inoltre, sarà il controller che, in qualità di coordinatore, dovrà sfruttare il model per fornire alla view tutti i dati che le servono, né più, né meno.

**Razor è il nome del view engine di ASP.NET Core MVC**, cioè del componente che si occupa di elaborare il i file .cshtml per generare il contenuto finale della pagina.

View di contenuto e view di layout

Le view si trovano all'interno della directory /Views del progetto. Servono a due scopi principali:

1. Le **view di contenuto** sono create per aiutare le singole action a produrre dinamicamente del contenuto HTML. Per questo, se vogliamo seguire la convenzione di nomi di ASP.NET Core MVC, dovremmo dare alla view **lo stesso nome dell'action** e inserirla in una **sottodirectory che si chiama come il controller**. Ad esempio, la view relativa all'action Index del CoursesController è preferibile che sia creata in /Views/Courses/Index.cshtml;
2. Le **view di layout** sono anch'esse dei file con estensione .cshtml e contengono gli elementi comuni dell'applicazione web come l'header, il menu di navigazione, footer e così via. Per rispettare la convenzione di nomi, è preferibile creare una view di layout in /Views/Shared/\_Layout.cshtml. Possiamo avere più di una view di layout, quindi non è obbligatorio rispettare questo nome. Anche **l'uso dell'underscore è consigliato** (ma non obbligatorio) e serve soltanto a far capire che il contenuto di questa view è riutilizzato e condiviso da altre view, come indica anche la directory Shared in cui si trova.

**Ogni view di contenuto può scegliere quale view di layout usare**. È sufficiente mettere in cima al file questa istruzione.

1. @{
2. Layout = "\_Layout";
3. }

Per evitare di ripetere questa istruzione in tutte le view di contenuto, la possiamo spostare all'interno di un file /Views/\_ViewStart.cshtml, che verrà eseguito prima che ogni view di contenuto sia elaborata.  
  
All'interno della view di layout, aggiungiamo invece un'istruzione @RenderBody() per indicare il punto in cui sarà incastonato l'output HTML prodotto dalla view di contenuto.

1. <html>
2. <head><!-- Contenuto della head --></head>
3. <body>
4. <header><!-- Contenuto dell'header --></header>
5. @RenderBody()
6. <footer><!-- Contenuto del footer --></footer>
7. </body>
8. </html>

Sezioni aggiuntive nella view di layout

Una view di layout può anche definire delle **sezioni aggiuntive**. Un esempio tipico è quello di una sezione chiamata "Scripts" in cui la view di contenuto potrà inserire del codice JavaScript, che è buona pratica inserire subito prima del tag di chiusura </body>. Ecco un esempio: nella view di layout usiamo @RenderSection("Scripts", required: false) per indicare che lì saranno incastonati i contenuti della sezione "Scripts".

1. <html>
2. <head><!-- Contenuto della head --></head>
3. <body>
4. <header><!-- Contenuto dell'header --></header>
5. @RenderBody()
6. <footer><!-- Contenuto del footer --></footer>
7. @RenderSection("Script", required: false)
8. </body>
9. </html>

E poi, nella view di contenuto, definiamo il contenuto della sezione "Scripts" all'interno di un blocco @section Scripts{...}. Definire questo contenuto è opzionale, dato che l'argomento required era stato impostato su false.

1. <h1>Elenco dei corsi</h1>
2. <p>Contenuto della pagina di elenco dei corsi</p>
3. @section Scripts {
4. <script>alert("Sono la pagina di elenco dei corsi!");</script>
5. }

Possiamo definire **quante sezioni preferiamo, con nomi qualsiasi**.

I tag helper

I tag helper **sono all'apparenza dei normalissimi tag** come a, form o input ma possiedono degli attributi aggiuntivi che ci **aiutano a produrre il contenuto HTML** nella view.

Ad esempio, con il tag helper a (anche chiamato tag helper *anchor*) possiamo usare gli attributi asp-action, asp-controller o asp-page per **creare un link indicando il nome dell'action, del controller o della Razor Page**, anziché andare a valorizzare il suo attributo href. In questo modo riusciamo a esprimere meglio il nostro intento: ecco un esempio in cui creiamo un link all'action Detail del CoursesController fornendo l'id 5.

1. <a asp-action="Detail" asp-controller="Courses" asp-route-id="5">Vai al dettaglio del corso 5</a>

I tag helper **riducono l'uso del codice C#** nella view e questo migliora la leggibilità del codice HTML. Così possiamo anche **coinvolgere i nostri colleghi designer** che, pur non avendo competenze in C#, magari sono molto abili a impaginare in HTML e perciò ci possono dare una mano. Dovremo giusto chiedergli di fare attenzione a non eliminare gli attributi che hanno il prefisso asp-.

Nel corso delle lezioni incontreremo vari tag helper e scopriremo i vantaggi che offrono. Poi, ne costruiremo uno personalizzato per sfruttare al massimo le loro potenzialità!

Usare librerie per lo sviluppo client come Bootstrap e FontAwesome

Progettare una UI usabile e gradevole per l'utente non è un compito semplice: ci sono professionisti specializzati in questo settore che possono aiutarci a realizzare un'applicazione di ottima qualità. Noi, come sviluppatori, il minimo che possiamo fare è procurarci delle librerie che ci guidano nella giusta direzione:

* [Bootstrap](https://getbootstrap.com/) offre un insieme di componenti e regole CSS che **migliora l'aspetto grafico e la fruibilità dei contenuti**. Il suo sistema di layout ci permette di creare pagine *responsive*, che si riorganizzano per adattarsi alla risoluzione dello schermo;
* [FontAwesome](https://fontawesome.com/) è un webfont che contiene tante **icone vettoriali** che possiamo usare nei nostri menu e sui nostri bottoni per comunicare meglio il loro scopo.

Per imparare a usare queste librerie possiamo consultare i rispettivi siti web, dove è riportata la documentazione e gli esempi. Per aggiungerle alla nostra applicazione, possiamo seguire uno o entrambi questi approcci:

* **Referenziarle da un CDN** come [cdnjs](https://cdnjs.com/) o [unpkg](https://unpkg.com/). I CDN aiutano le nostre pagine a caricarsi più in fretta perché l'utente potrebbe avere già in cache quella libreria, se in precedenza aveva visitato un altro sito che ne faceva uso;
* **Ottenerle con LibMan**. LibMan è uno strumento da riga di comando che si installa con dotnet tool install -g Microsoft.Web.LibraryManager.Cli. Ci permette di scaricare i file delle librerie nel nostro progetto con comandi come libman install font-awesome. In questo modo, tutti i file vengono scaricati localmente e serviti dall'applicazione ASP.NET Core. Perciò, non dipendiamo da CDN di terze parti.

Possiamo sfruttare entrambi gli approcci usando il tag helper <environment>: mentre siamo in sviluppo usiamo le librerie che abbiamo ottenuto grazie a LibMan, così non sarà necessaria la connessione a internet. Completato lo sviluppo, allora sfrutteremo i CDN. Ecco un esempio di come referenziare i file .css delle due librerie nella view di layout.

1. <environment include="Development">
2. <!-- In sviluppo usiamo file locali -->
3. <link rel="stylesheet" href="~/lib/font-awesome/css/font-awesome.min.css" />
4. <link rel="stylesheet" href="~/lib/twitter-bootstrap/css/bootstrap.min.css" />
5. </environment>
6. <environment exclude="Development">
7. <!-- In tutti gli altri ambienti usiamo i CDN -->
8. <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" />
9. <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css" />
10. </environment>

Grazie agli attributi include e exclude del tag helper environment possiamo mostrare o nascondere dei blocchi di codice HTML in base al nome dell'ambiente.

Essere più produttivi con live reload e dotnet watch run

Quando modifichiamo i contenuti della view, siamo molto più produttivi se il browser si aggiorna automaticamente non appena salviamo. Questa funzionalità si chiama **live reload** e la possiamo avere installando l'estensione [Browser Sync](https://marketplace.visualstudio.com/items?itemName=jasonlhy.vscode-browser-sync) di Visual Studio Code. Dopo averla installata, aggiungiamo questo pezzetto di configurazione al file .vscode/settings.json, che permetterà all'estensione di reagire ai cambiamenti ai file .cshtml e a ogni file statico situato in wwwroot.

1. "browserSync.config": {
2. "proxy": "localhost:5000",
3. "files": ["Views/\*\*/\*.cshtml", "wwwroot/\*\*/\*"]
4. }

Poi, da Visual Studio Code, avviamo l'applicazione e poi digitiamo proxy nella command palette per trovare i comandi di avvio dell'estensione Browser Sync.

**Non è necessario ricompilare l'intera applicazione ogni volta che apportiamo una modifica a una view.** Infatti, le view vengono dinamicamente ricompilate se vengono modificate. Per il codice C# invece è diverso: se apportiamo **una modifica a un controller, l'intera applicazione deve essere ricompilata**. Per ridurre questi tempi di attesa, possiamo lanciare l'applicazione con il comando dotnet watch run che scatenerà la ricompilazione ad ogni modifica su un file .cs. Così, in combinazione con il live reload, possiamo continuare a lavorare a progetto senza distrazioni!

Attenzione: quando usiamo il comando dotnet watch run **non possiamo avvalerci del debugger**.

Altre estensioni consigliate per Visual Studio Code

* [Abilitando Emmet](https://code.visualstudio.com/docs/editor/emmet), già incluso in Visual Studio Code, possiamo generare codice HTML molto velocemente;
* Con [Boostrap v4 Snippets](https://marketplace.visualstudio.com/items?itemName=Zaczero.bootstrap-v4-snippets) possiamo inserire i componenti di Bootstrap senza doverli copiare dalla documentazione;
* Con [Font Awesome Auto-complete & Preview](https://marketplace.visualstudio.com/items?itemName=Janne252.fontawesome-autocomplete) è più facile usare le icone di FontAwesome;
* Con [GitLens](https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens) abbiamo un valido supporto per esplorare la storia dei commit nel repository Git.