**Riepilogo della sezione**

Ricevere l'input dell'utente grazie al model binding

ASP.NET Core possiede un meccanismo chiamato **model binding** che si occupa di esaminare la richiesta HTTP per estrarre informazioni utili, così che poi possiamo usarle nelle action dei nostri controller. Si compone di due organi principali:

* I **value providers** si occupano di estrarre i valori grezzi, in formato stringa, dalle varie parti della richiesta come route parameters, query string, form data e intestazioni;
* I **model binders** si occupano invece di convertire quei valori grezzi nei tipi di dato che sono stati espressi come parametri dell'action.

Ecco per esempio una richiesta in cui è stata fornita una chiave query string page valorizzata su 5.

https://localhost:5001/Courses?page=5

Ed ecco l'action che gestirà tale richiesta, che riceverà il valore 5 dal suo parametro page di tipo int.

1. public ActionResult Index(int page)
2. {
3. //...
4. }

Senza che noi dovessimo far nulla, il valore 5 è stato recuperato dalla query string grazie a un *value provider* e poi convertito al tipo int da un *model binder*.

I *model binder* sono molto potenti e riescono a creare sia oggetti di tipo primitivo (come int e bool) ma anche **di tipo complesso**, come classi personalizzate create da noi. Quest'ultimo aspetto è molto importante, soprattutto quando realizzeremo maschere di modifica, perché ci permette di tenere coesi tanti valori in un unico oggetto.

Il meccanismo del model binding usa delle **convenzioni di nomi** ed è per questo che il valore fornito con una chiave querystring chiamata page può arrivare al parametro di un'action chiamato anch'esso page. Il confronto avviene in maniera *case-insensitive*.

In una richiesta possono esserci *collisioni di nomi* che possono portare ad **ambiguità**, come ad esempio una chiave query string chiamata page e un campo del form chiamato anch'esso page. In questo caso possiamo usare degli **attributi** per essere espliciti sulla fonte da usare per trarre il valore. Ecco un esempio in cui usiamo l'attributo FromQuery in corrispondenza del parametro page per indicare che il valore deve essere tratto dalla query string.

1. public ActionResult Index([FromQuery] int page)
2. {
3. //...
4. }

Ecco un elenco degli attributi che possiamo porre di fianco ai parametri dell'action:

* FromForm attinge dai valori inviati con un form (metodo POST);
* FromRoute dai parametri indicati nel route template (ad esempio l'id indicato in {controller}/{action}/{id});
* FromQuery dalle chiavi  query string;
* FromHeader dalle intestazioni della richiesta HTTP. È necessario usarlo se vogliamo attingere valori da lì;
* FromServices non attinge valori dalla richiesta ma dagli oggetti registrati per la dependency injection.

Creare un model binder personalizzato

Se non siamo soddisfatti del modo in cui i *model binder*  costruiscono i nostri oggetti, possiamo pur sempre crearne uno personalizzato. Si tratta semplicemente di costruire una classe come la seguente, che implementa IModelBinder e sfrutta i value provider per recuperare i valori dalla richiesta.

1. public class MyModelBinder : IModelBinder
2. {
3. public Task BindModelAsync(ModelBindingContext bindingContext)
4. {
5. //Recuperiamo un valore grazie ai value provider e lo convertiamo a intero
6. string page = Convert.ToInt32(bindingContext.ValueProvider.GetValue("page").FirstValue);
8. //Creiamo l'istanza del nostro oggetto fornendogli il valore
9. var inputModel = new MyObject(page);
11. //Impostiamo il risultato per notificare che la creazione è avvenuta con successo
12. bindingContext.Result = ModelBindingResult.Success(inputModel);
14. //Restituiamo un task completato
15. return Task.CompletedTask;
16. }
17. }

A questo punto poniamo l'attributo ModelBinder sul tipo di oggetto che vogliamo venga costruito dal nostro *model binder* personalizzato.

1. [ModelBinder(BinderType = typeof(MyModelBinder))]
2. public class MyObject
3. {
4. //...
5. }

Come gli altri componenti della nostra applicazione, anche i model binder personalizzati **supportano la dependency injection** e perciò possono manifestare dipendenze da altri servizi nel loro costruttore.

Consentire all'utente di inviare informazioni

Esistono vari modi per fare in modo che l'utente ci invii informazioni:

* Se deve semplicemente scegliere tra un insieme finito di possibilità possiamo creare dei link che rechino delle chiavi in query string. Per far questo usiamo il **tag helper a** nelle nostre view e impostiamo il suo attributo asp-route-\*:
  1. <a asp-controller="NomeController" asp-action="NomeAction" asp-route-page="1">Pagina 1</a>
* Se deve poter digitare un testo breve possiamo predisporre un **elemento input** all'interno di un **form con attributo method="get"**. In questo modo il testo digitato dall'utente apparirà in query string e ciò gli consentirà di condividere l'URL sui social o di aggiungerlo ai preferiti per tornarci in seguito:
  1. <form asp-controller="NomeController" asp-action="NomeAction" method="get">
  2. <input type="text" name="search" placeholder="Cerca...">
  3. <button>Invia</button>
  4. </form>
* Se deve compilare svariate informazioni, potenzialmente corpose, allora creiamo un **form con method="post"**. Usiamo questa tecnica anche se non vogliamo che le informazioni digitate dall'utente appaiano nell'URL.

Implementare ricerca, paginazione e ordinamento

Per rendere più fruibili i nostri elenchi di dati, è consigliabile consentire all'utente di digitare un testo da cercare oppure di selezionare un criterio di ordinamento e il numero di pagina da visualizzare.

Grazie al *model binding* potremo ricevere i valori inviati dall'utente come parametri delle nostre action. A quel punto, li forniremo ai servizi applicativi affinché li integrino nelle loro query SQL o LINQ così che possano restituire risultati pertinenti.

Se stiamo usando query **SQL con ADO.NET**:

* Per la **ricerca** ci basta usare la clausola WHERE *NomeCampo* LIKE '%testo%';
* Per la **paginazione** aggiungiamo in fondo alla query le clausole LIMIT *daPrendere* OFFSET *daSaltare* (per Sqlite, MySQL, PostgreSQL) o OFFSET *daSaltare* ROWS FETCH NEXT *daPrendere* ROWS ONLY per (SqlServer o Oracle);
* Per l'**ordinamento**, dopo la WHERE usiamo la clausola ORDER BY *NomeCampo* ASC (senso ascendente) oppure ORDER BY *NomeCampo* DESC (senso discendente). La parola ASC è la predefinita e può essere omessa. Se vogliamo ordinare per più di un campo, ci basterà separarli con la virgola come in ORDER BY *NomeCampo1*DESC*, NomeCampo2*.

Se stiamo usando query **LINQ con Entity Framework Core**:

* Per la **ricerca** aggiungiamo alla query LINQ l'extension method Where;
* Per la **paginazione** usiamo  Skip e Take;
* Per l'**ordinamento** usiamo OrderBy o OrderByDescending. Se vogliamo ordinare per più di un campo, aggiungiamo di seguito anche gli extension method ThenBy e ThenByDescending.