04/04/20 - Versione 1

Blazor e il pattern MVVM

Blazor è un nuovo framework che permette di creare single page application utilizzando l’ecosistema .Net Core invece di javascript.

Il pattern Model-View-ViewModel, anche se nato per essere utilizzato nell’ambito desktop, può essere utilizzato efficacemente per scrivere applicazioni con questo, anche se non viene fornito di default alcun tipo di supporto nativo alla sua applicazione.

Questa mancanza non sarà nemmeno probabilmente colmata nelle future release: infatti gli stessi creatori di Blazor hanno evidenziato come questo sia stato progettato in modo da non favorire alcun pattern di sviluppo particolare, in modo da favorirne la massima flessibilità nel suo utilizzo.

Riguardo a MVVM gli stessi hanno anche affermato che “Blazor nel futuro non favorirà in alcun modo l’adozione di MVVM ma nemmeno ne bloccherà o svantaggerà in alcun modo l’utilizzo” (linkografia 1).

In questo articolo analizzeremo inizialmente le basi fondanti del pattern, quindi svilupperemo una semplice web-app che ne utilizza le peculiarità.

Solo successivamente descriveremo lo stesso progetto però adottando il framework BlazorMVVM, che permette di semplificare l’organizzazione del codice.

Vedremo anche come l’utilizzo del pattern MVVM in Blazor risulta essere facile e intuitivo, pur senza avere l’ausilio di strumenti nativi da parte del framework, e permette di ottenere diversi vantaggi nell’organizzazione del codice e delle logiche di funzionamento.

Descrizione del pattern

Prima di descrivere come implementare il pattern in un progetto Blazor descriveremo qui le basi teoriche che sottendono il suo funzionamento.

Il pattern Model–view–viewmodel (MVVM) è stato introdotto da Microsoft per favorire lo sviluppo in ambito Windows Presentation Foundation (WPF), anche se riprende basi teoriche precedenti.

Ad oggi viene estensivamente usato non solo in WPF, ma anche con molti altri framework, per esempio Xamarin.

Il suo intento è quello di separare il codice in modo tale creare moduli distinti per le funzionalità dedicate alla user-interface e alla logica di business.

Nell’ambito del pattern vengono identificati i seguenti componenti costitutivi.

* Models: sono le classi che contengono la struttura dei dati trattati. Si tratta in generale di classi POCO e structs.
* Views: Sono i moduli delegati a visualizzare la user-interface. Nell’ambito Blazor si tratta dei componenti razor, invece in WPF/Xamarin dei file Xaml che rappresentando la user-interface. In questa categoria appartengono anche i relativi code-behind che equipaggiano le user-interface poiché, essendo intimamente collegata alla sua Views, sono da considerarsi, nell’ambito del pattern, un tutt’uno con questa.
* ViewModel: E’ la classe che è delegata a trasformare le informazioni contenuti nei modelli in modo tale da essere visualizzate nella View.

<<Figura 01 – da sistemare perché questa è rubata da- sito M$>>

La View-Models può essere associata alla Views impostando il data-context di quest’ultima a un’istanza della classe.

Fatto questo è possibile associare i controlli contenuti alle varie proprietà esposte dalla classe ViewModel tramite un’operazione di databinding.

<<Completare con qualche altro paragrafo che dettaglia il pattern>>

Vantaggi e svantaggi relativi all’utilizzo del pattern MVVM

Possiamo evidenziare due motivi principali per adottare il pattern MVVM nell’ambito dello sviluppo Blazor.

* Riutilizzo di codice già esistente con altri progetti dotati di user interface (Es.: WPF/Xamarin)
* Possibilità di utilizzare gli unit-test per verificare il comportamento del ViewModel

Tra gli svantaggi principali che si possono apprezzare è che per progetti piccoli, o dotati di poche funzionalità, l’adozione del MVVM porta una complicazione nell’organizzazione del codice, a fronte di vantaggi limitati per il tipo di progetto.

Una prima implementazione

Per seguire un esempio pratico nel seguito immaginiamo di voler realizzare una semplice ToDoList, in cui ogni item è costituito da una descrizione dell’attività e da una data massima entro la quale debba essere svolta.

Dal punto di vista dell’aspetto il codice sarà organizzato in modo tale da avere una pagina principale che fungerà da contenitore, e che ospiterà al suo interno due componenti figli: uno che visualizzerà la lista dei ToDo e l'altro delegato esclusivamente all'inserimento di nuovi elementi.

<<Immagine Pagina con evidenza della disposizione dei componenti>>

Questi componenti condivideranno la stessa lista di TodoItem conservata all’interno del ViewModel.

Per iniziare creiamo un nuovo progetto Blazor Server App, che nomineremo BlazorToDoListMVVM.

Ottenuto questo creiamo all’interno di questo due cartelle: una chiamata Models, delegata a contenere le classi POCO che rappresentano le informazioni trattate, e un’altra chiamata ViewModels, che invece conterrà le classi di ViewModel utilizzate nel progetto.

Questa struttura ci permetterà di organizzare meglio le classi utilizzate.

Aggiungiamo ora la classe ToDoItem, che conterrà le informazioni di ogni ToDo.

Models/ToDoItem.cs

|  |
| --- |
| namespace BlazorToDoListMVVM.Models  {  public class ToDoItem  {  [Required]  public string Description { get; set; }  public DateTime? DueDate { get; set; }  }  } |

Successivamente creiamo la classe ViewModel, che sarà utilizzata da tutti i componenti coinvolti nel progetto.

ViewModels/ToDoItemsListViewModel.cs

|  |
| --- |
| namespace BlazorToDoListMVVM.ViewModels  {  public class ToDoItemsListViewModel : INotifyPropertyChanged  {  public ToDoItemsListViewModel()  {    }  public event PropertyChangedEventHandler PropertyChanged;  public void AddToDoItem(ToDoItem todoitem)  {  \_ToDoItemList.Add(todoitem);  OnPropertyChanged(nameof(ToDoItemList));  }  private string \_Title = string.Empty;  public string Title  {  get => \_Title;  set  {  \_Title = value;  OnPropertyChanged();  }  }  private List<ToDoItem> \_ToDoItemList = new List<ToDoItem>();  public List<ToDoItem> ToDoItemList  {  get => \_ToDoItemList;  set  {  \_ToDoItemList = value;  OnPropertyChanged();  }  }  private void OnPropertyChanged([CallerMemberName] string propertyName = null)  {  if (PropertyChanged == null)  return;  PropertyChanged.Invoke(this, new PropertyChangedEventArgs(propertyName));  }  }  } |

La classe espone verso l’esterno una lista di TodoItem, nonché una stringa che rappresenta il titolo da assegnare alla pagina principale.

Oltre a questo, implementa anche l’interfaccia INotifyPropertyChanged: questa prevede l’esistenza di un solo evento, chiamato PropertyChanged.

|  |
| --- |
| public interface INotifyPropertyChanged  {  event PropertyChangedEventHandler PropertyChanged;  } |

Questa interfaccia è stata creata originariamente per permettere alla ViewModel di notificare alla user-inteface che è stato modificato un valore di una sua proprietà proposta verso l’esterno: l’argomento fornito dall’evento riporta proprio il nome di questa.

Quindi per esempio nell’ambito di WPF/Xamarin il motore grafico che si occupa di disegnare la maschera sottoscrive a questo evento in modo tale da essere avvisata ogni qualvolta un valore tra quelli esposti dal ViewModel varia, e pertanto sarà in grado di eseguire le azioni necessarie per aggiornare la visualizzazione del controllo associato.

Per questo motivo all’interno della classe occorre prevedere un meccanismo che deve fare scattare questo evento nel momento opportuno, cioè alla modifica del valore di una proprietà: questa funzionalità è implementata all’interno del metodo OnPropertyChanged.

Se si usasse WPF/Xamarin la sottoscrizione all’evento da parte della user-interface avverrebbe automaticamente quando si associa l’istanza della classe ViewModel al relativo data-context.

Questa possibilità però, non è disponibile in Blazor: infatti qui non esiste il concetto di data-context associato alla user-interface, e pertanto è necessario sottoscrivere esplicitamente l’evento. Inoltre, come detto all’inizio, il framework non mette a disposizione alcun strumento particolare per favorire l’adozione di alcun pattern di programmazione, e quindi nemmeno per MVVM.

Modifichiamo, quindi, la pagina Index.razor come nel seguito.

Pages/Index.razor

|  |
| --- |
| @page "/"  @inject ToDoItemsListViewModel ViewModel  <h1>@ViewModel.Title</h1>  <ToDoFormComponent ViewModel="@ViewModel" />  <ToDoListComponent ViewModel="@ViewModel" />  @code {  protected override void OnInitialized()  {  base.OnInitialized();  ViewModel.Title="Lista ToDoItem";  ViewModel.PropertyChanged+=(o, e) => InvokeAsync(() =>  {  StateHasChanged();  });;  }  } |

Affinché la View (cioè la pagina razor) venga correttamente aggiornata occorre sottoscrivere l’evento PropertyChanged, associandolo al metodo di Blazor StateHasChanged: questo avviene all’interno di OnInitialized, che viene richiamato quando si crea la pagina.

Occorre osservare che StateHasChanged, richiamato dalla pagina principale, avrà come effetto quello forzare un refresh di quanto rappresentato da tutti i componenti discendenti: per tale motivo su questi non sarà necessario eseguire alcuna implementazione per sottoscrivere a questo evento.

Nel caso in analisi non si considera in alcun modo il parametro dell’evento PropertyChanged, che riporta originariamente il nome della proprietà variata: semplicemente ogni volta che scatta l’evento viene richiamato StateHasChanged, e questo avrà come effetto quello di ridisegnare l’intera visualizzazione.

Per permettere all’injection di funzionare nel modo corretto, ovvero di fornire l’istanza della classe ViewModel a tutte le classi richiedenti, occorre agire in startup, registrando la classe come Scoped, poiché vogliamo che l'istanza abbia un ciclo di vita legato alla sessione dell'utente.

Modifichiamo la classi di startup come nel seguito.

Startup.cs

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  ...  services.AddScoped<ToDoItemsListViewModel>();  ...  } |

La stessa istanza della classe ViewModel, una volta ottenuta dalla pagina Index.razor, viene propagata ai componenti discendenti che costituiscono la pagina, tramite parametro nominato.

Spostiamoci ora ad analizzare il primo dei due componenti: quello delegato a visualizzare la lista dei todoitem.

Shared/ToDoListComponent.razor

|  |
| --- |
| <h3>Lista ToDo</h3>  @if (ViewModel.ToDoItemList.Count() <= 0)  {  <p><em>Nessun item</em></p>  }  else  {  <table class="table">  <thead>  <tr>  <th>Description</th>  <th>DueDate</th>  </tr>  </thead>  <tbody>  @foreach (var todoitem in ViewModel.ToDoItemList)  {  <tr>  <td>@todoitem.Description</td>  <td>  @(todoitem.DueDate?.ToShortDateString() ?? string.Empty)  </td>  </tr>  }  </tbody>  </table>  }  @code {  [Parameter]  public ToDoItemsListViewModel ViewModel { get; set; }  } |

Questo componente disegna una semplice tabella che ospita tutti gli item contenuti nella lista.

Occorre osservare che in ambito WPF/Xamarin all’interno della ViewModel si sarebbe definita la lista dei Todo come tipo ObservableCollection, in modo tale che quando la si associata nella user-interface ad un componente tipo il ListView, quest’ultima avrebbe sempre visualizzato il corretto numero di item anche in caso di modifiche nella lista stessa.

Infatti, il meccanismo proposto dall’oggetto ObservableCollection funziona analogamente a quanto visto con la ViewModel e l’emissione della proprietà PropertyChanged, esponendo verso l’esterno un evento, questa volta chiamato CollectionChanged, in grado di scattare automaticamente ogni qualvolta venga aggiunto o rimosso un item al suo interno.

Grazie a questo meccanismo la ListView è in grado di sottoscrivere a questo evento e quindi modificare la sua visualizzazione in sincronia con gli item contenuti nella lista: anche in questo caso tutto avviene automaticamente semplicemente associando la lista al controllo.

In Blazor, però, non si ha la disponibilità di un controllo come questo, e pertanto si rende necessario far scattare l’evento PropertyChanged, che ridisegna l’intera visualizzazione grazie a StateHasChanged, in ogni metodo della ViewModel che modifica il numero degli item della lista: nel nostro caso specifico all’interno del metodo AddToDoItem e anche all’interno della proprietà ToDoItemList, nel verbo set.

L'altro componente che concorre a comporre la pagina è delegato a contenere la form di inserimento di nuovi elementi nella todolist.

Shared/ToDoFormComponent.razor

|  |
| --- |
| <h3>Nuovo ToDo</h3>  <EditForm Model="@Newtodoitem" OnValidSubmit="@CreaTodo">  <DataAnnotationsValidator />  <ValidationSummary />  <p>  <div class="form-field">  <label for="descrizione">Descrizione</label>  <InputText id="descrizione" @bind-Value=@Newtodoitem.Description />  </div>  </p>  <p>  <div class="form-field">  <label for="duedate">Due Date</label>  <InputDate id="duedate" @bind-Value=@Newtodoitem.DueDate ParsingErrorMessage="Inserire un data valida" />  </div>  </p>  <button type="submit">Salva</button>  </EditForm>  @code {  [Parameter]  public ToDoItemsListViewModel ViewModel { get; set; }  private ToDoItem Newtodoitem = new ToDoItem();  void CreaTodo()  {  ViewModel.AddToDoItem(Newtodoitem);  Newtodoitem = new ToDoItem();  }  } |

Il componente utilizza al suo interno il controllo Blazor EditForm, che nella sostanza permette di editare le proprietà esposte dall’istanza della classe ToDoItem.

Il tag DataAnnotationsValidator configura il controllo in modo che validi i dati inseriti rispetto agli attributi assegnati all’interno delle classi POCO: quindi nel nostro caso verifica che la proprietà Descrizione disponga di qualche valore.

Invece il tag ValidationSummary permette di visualizzare i messaggi relativi ad errori di validazione: solo nel caso in cui la validazione abbia esito positivo il controllo alla pressione del tasto salva richiamerà l’handler CreaToDo.

Riutilizzo della classe ViewModel in Xamarin/WPF: ICommand e ObservableCollection

Immaginiamo, ora, di voler utilizzare lo stesso ViewModel analizzato sinora sia in ambito Blazor che in un progetto WPF/Xamarin, con l’intento di favorire il riutilizzo del codice.

Per ottenere questo occorre introdurre dei correttivi alla classe ViewModel in analisi in modo tale da renderla correttamente compatibile con entrambe le tecnologie.

Per iniziare dobbiamo occuparci di come gestire la pressione del tasto salva all’interno del componente ToDoFormCompoent.razor.

Nella versione sinora in analisi per associare l’esecuzione del metodo AddToDoItem alla pressione del tasto salva si è utilizzato l’handler del controllo associato a OnValidSubmit, e direttamente da lì si è richiamato esplicitamente il metodo.

Questo è stato possibile grazie al fatto che il metodo AddToDoItem è stato caratterizzato come pubblico.

Se si volesse riutilizzare la stessa classe ViewModel anche in un progetto WPF/Xamarin occorre introdurre per questa parte un refactoring: infatti in questo ambito l’azionamento di un metodo alla pressione di un tasto avviene in modo diverso.

Infatti, in questo caso si espone dal ViewModel un oggetto che implementa l’interfaccia ICommand: il bottone salva si configura in modo tale da essere messo in binding con tale proprietà.

L’interfaccia ICommand definisce i seguenti due metodi e un evento.

|  |
| --- |
| public interface ICommand  {  void Execute(object parameter);  bool CanExecute(object parameter);  event EventHandler CanExecuteChanged;  } |

Execute è l’handler che viene richiamato alla pressione del tasto: CanExecute invece è un metodo che deve restituire un booleano per indicare se è possibile o meno richiamare Execute.

Infine, l’evento CanExecuteChanged deve essere emesso tutte le volte che il valore di CanExecute varia.

Lo scopo di CanExecute e CanExecuteChanged è quello di permettere alla user-interface WPF/Xamarin di ridisegnare il bottone associato nel modo corretto: infatti quando si esegue il binding di un’istanza di questa classe a un bottone la logica di funzionamento sottoscriverà automaticamente all’evento e introdurrà le azioni necessarie.

Pertanto, per se si rende necessario disabilitare il bottone associato, per le logiche di funzionamento implementate all’interno della ViewModel, è sufficiente emettere l’evento CanExecuteChanged e far restituire a CanExecute il valore false: in questo modo si ottiene anche che il bottone sia mostrato con il classico aspetto di disabilitato.

Occorre rammentare che la ViewModel non ha alcun accesso ai controlli grafici, e questo meccanismo è l’unico possibile per introdurre la funzionalità.

Quindi per rendere compatibile il ViewModel occorre modificare il modo con cui viene richiamato il metodo AddToDoItem, in modo tale che questa sia compatibile in entrambi gli ambiti, e quindi introducendo l’utilizzo di un oggetto che implementi l’interfaccia ICommand.

Partiamo modificando la solution appena creata (tenendone una copia, che utilizzeremo nell’ultima parte dell’articolo) e aggiungiamo un nuovo progetto di tipo Class Library (.Net Standard) che chiameremo BlazorToDoListMVVMLib.

Questo progetto sarà delegato a contenere il modello e la ViewModel, e quindi potrà essere utilizzata da un eventuale progetto XPF o Xamarin, oltre che dal progetto Blazor.

Impostiamo, ora, il riferimento al progetto appena creato dal progetto BlazorToDoListMVVM: quindi possiamo spostare le due cartelle ViewModels e Models, comprensive del loro contenuto, all’interno del progetto BlazorToDoListMVVMLib.

Non modificheremo in alcun modo la classe POCO ToDoItem: semplicemente variamo il namespace per renderlo congruente con il progetto.

Models\ToDoItem.cs

|  |
| --- |
| namespace BlazorToDoListMVVMLib.Models  {  public class ToDoItem  {  [Required]  public string Description { get; set; }  public DateTime? DueDate { get; set; }  }  } |

Per via dell’attributo required che correda la proprietà Description sarà necessario installare nel progetto il nuget package System.ComponentModel.Annotations, che di default non è presente nei progetti di tipo Class Library (.Net Standard).

La classe ViewModels invece sarà modificata come nel seguito.

|  |
| --- |
| namespace BlazorToDoListMVVMLib.ViewModels  {  public class ToDoItemsListViewModel : INotifyPropertyChanged  {  public ToDoItemsListViewModel()  {  SalvaCommand = new DelegateCommand((x) => AddToDoItem(x), x => !IsBusy);  }  public DelegateCommand SalvaCommand { get; set; }  public bool IsBusy { get; set; }  public event PropertyChangedEventHandler PropertyChanged;  private void AddToDoItem(object todoitem)  {  IsBusy = true;  ToDoItem toDo = todoitem as ToDoItem;  \_ToDoItemList.Add(toDo);  OnPropertyChanged(nameof(ToDoItemList));  IsBusy = false;  }  private string \_Title = string.Empty;  public string Title  {  get => \_Title;  set  {  \_Title = value;  OnPropertyChanged();  }  }  private ObservableCollection<ToDoItem> \_ToDoItemList = new ObservableCollection<ToDoItem>();  public ObservableCollection<ToDoItem> ToDoItemList  {  get => \_ToDoItemList;  set  {  \_ToDoItemList = value;  OnPropertyChanged();  }  }  public void OnPropertyChanged([CallerMemberName] string propertyName = null)  {  if (PropertyChanged == null)  return;  PropertyChanged.Invoke(this, new PropertyChangedEventArgs(propertyName));  }  }  } |

Con questo refactory viene proposto verso l’esterno la proprietà SalvaCommand, di tipo DelegateCommand: questa classe implementa l’interfaccia ICommand e sarà questa proprietà permetterà di richiamare dall’esterno il metodo AddToDoItem, che a questo punto può essere modificato con visibilità privata.

Service/DelegateCommand.cs

|  |
| --- |
| namespace BlazorToDoListMVVMLib.Service  {  public class DelegateCommand : ICommand  {  private readonly Action<object> \_action;  private readonly Func<object, bool> \_condition;  public DelegateCommand(Action<object> action, Func<object, bool> condition)  {  \_action = action;  \_condition = condition;  }  public void Execute(object parameter)  {  if (CanExecute(parameter))  \_action.Invoke(parameter);  }  public bool CanExecute(object parameter)  {  return \_condition == null || \_condition.Invoke(parameter);  }  public event EventHandler CanExecuteChanged;  public void RaiseCanExecuteChanged()  {  if (CanExecuteChanged != null)  CanExecuteChanged(this, EventArgs.Empty);  }  }  } |

Il costruttore della classe DelegateCommand accetta una Action, che è il delegate al metodo del ViewModel da richiamare, e una function, che rappresenta la logica implementativa di CanExecute.

Completa la classe un metodo pubblico RaiseCanExecuteChanged che risulta necessario per emettere l’evento CanExecuteChanged dall’interno della ViewModel.

Il refactoring introdotto nella gestione dell’aggiunta degli item alla lista pone come obbligo di modificare il componente ToDoFormComponent: infatti Blazor non è in alcun modo in grado di utilizzare al suo interno un oggetto ICommand, e tantomeno di gestirne il binding com un bottone. Sarà necessario richiamare direttamente il metodo esposto.

Shared/ToDoFormComponent.razor

|  |
| --- |
| <h3>Nuovo ToDo</h3>  <EditForm Model="@Newtodoitem" OnValidSubmit="@CreaTodo">  <DataAnnotationsValidator />  <ValidationSummary />  <p>  <div class="form-field">  <label for="descrizione">Descrizione</label>  <InputText id="descrizione" @bind-Value=@Newtodoitem.Description />  </div>  </p>  <p>  <div class="form-field">  <label for="duedate">Due Date</label>  <InputDate id="duedate" @bind-Value=@Newtodoitem.DueDate ParsingErrorMessage="Inserire un data valida" />  </div>  </p>  <button type="submit">Salva</button>  </EditForm>  @code {  [Parameter]  public ToDoItemsListViewModel ViewModel { get; set; }  private ToDoItem Newtodoitem = new ToDoItem();  void CreaTodo()  {  if (ViewModel.SalvaCommand.CanExecute(null))  ViewModel.SalvaCommand.Execute(Newtodoitem);  Newtodoitem = new ToDoItem();  }  } |

Prima di richiamare il metodo Execute, che a sua volta ha associato come Action il metodo AddToDoItem, si testa se ne è possibile l’esecuzione utilizzando CanExecute.

In alcun modo si utilizza l’evento CanExecuteChanged in Blazor: pertanto per disabilitare eventualmente un bottone occorrerà agire in altro modo oppure occorrerà sottoscrivere manualmente all’evento.

Per completare la descrizione nel ViewModel è possibile osservare che la lista dei ToDo con questa versione viene esposta verso l’esterno tramite un oggetto ObservableCollection: questo per essere consistenti con quanto detto in precedenza e quindi offrire la massima compatibilità con WPF/Xamarin.

Siccome ObservableCollection implementa a sua volta l’interfaccia IList, la stessa dell’oggetto List usato in precedenza, e siccome Blazor non sottoscrive in alcun modo all’evento CollectionChanged di questa, per questa parte non occorre eseguire alcuna modifica nel componente Blazor ToDoListComponent, che è delegato a rappresentare la lista degli item.

Con queste modifiche la stessa classe ViewModel può essere utilizzata efficacemente sia nel progetto Bazor visto prima, che in un progetto WPF/Xamarin.

Utilizzo di BlazorMVVM

Come visto per condividere l’utilizzo di una classe ViewModel sia in ambito Blazor che in WPF/Xamarin occorrono alcune modifiche per poter conciliare le esigenze di entrambe le piattaforme.

Se però non si avesse la necessità di condividere questa classe con altri tipi di progetto, bensì la si volesse usare esclusivamente nell’ambito Blazor, è possibile utilizzare una framework scritto allo scopo: MvvmBlazor.

Per esporne l’utilizzo partiamo dalla prima versione del progetto, e aggiungiamo il pacchetto nuget MvvmBlazor.

Per inizializzare la libreria all’interno del dependecy injection fornito da .Net Core occorre modificare anche la classe startup del progetto.

Startup.cs

|  |
| --- |
| public class Startup  {  public Startup(IConfiguration configuration)  {  Configuration = configuration;  }  public IConfiguration Configuration { get; }    public void ConfigureServices(IServiceCollection services)  {  …...  services.AddMvvm();  …...  } |

Fatto questo possiamo iniziare a utilizzarne le funzionalità. Per permettere il corretto funzionamento occorre che le classi di ViewModel derivino da ViewModelBase, una classe messa a disposizione da MvvmBlazor.

ViewModels/ToDoItemsListViewModel.cs

|  |
| --- |
| public class ToDoItemsListViewModel : ViewModelBase  {  public ToDoItemsListViewModel()  {    }  public void AddToDoItem(ToDoItem)  {  ToDoItemList.Add(todoitem);  }  private string \_Title = string.Empty;  public string Title  {  get => \_Title;  set => Set(ref \_Title, value);    }  public ObservableCollection<ToDoItem> ToDoItemList { get; } = new ObservableCollection<ToDoItem>();    public override void OnInitialized()  {  base.OnInitialized();  Title = "Lista ToDoItem";    }  } |

ViewModelBase implementa al suo interno InotifyPropertyChanged, per cui non è più necessario nella classe ViewModel implementare in alcun modo esplicitamente questa interfaccia.

Per emettere l’evento PropertyChanged è possibile richiamare il metodo Set messo a disposizione dalla classe ViewModelBase, che si occupa di emettere l’evento nel modo corretto.

Per quanto riguarda la lista ToDoItemList questa volta può essere esposta con l’utilizzo dell’oggetto ObservableCollection: infatti ora il framework è in grado di sottoscrivere all’evento CollectionChanged e gestirlo nel modo corretto richiamando al suo interno StateHasChanged.

Pertanto, non sarà più necessario emettere esplicitamente all’interno della ViewModel l’evento PropertyChanged sia da dentro la proprietà ToDoItemList che dal metodo AddToDoItem.

Anche i componenti coinvolti devono essere modificati affinché tutto funzioni correttamente: in particolare devono essere modificati tutti in modo tale da discendere tutti da MvvmComponentBase<T>, dove T è il tipo che si riferisce alla classe ViewModel.

Pages\Index.razor

|  |
| --- |
| @page "/"  @inherits MvvmComponentBase<ToDoItemsListViewModel>  <h1>@Bind(x => x.Title) </h1>  <ToDoFormComponent />  <ToDoListComponent /> |

Il binding con la proprietà Title viene favorito con l’utilizzo del metodo Bind. Con questa versione non è più necessario sottoscrivere esplicitamente all’evento PropertyChanged: ora la libreria farà l’operazione autonomamente.

Shared/ToDoListComponent.razor

|  |
| --- |
| @inherits MvvmComponentBase<ToDoItemsListViewModel>  <h3>Lista ToDo</h3>  @if (@Bind(x => x.ToDoItemList).Count() <= 0)  {  <p><em>Nessun item</em></p>  }  else  {  <table class="table">  <thead>  <tr>  <th>Description</th>  <th>DueDate</th>  </tr>  </thead>  <tbody>  @foreach (var todoitem in @Bind(x => x.ToDoItemList))  {  <tr>  <td>@todoitem.Description</td>  <td>  @(todoitem.DueDate?.ToShortDateString() ?? string.Empty)  </td>  </tr>  }  </tbody>  </table>  } |

L’unica modifica sostanziale eseguita in questo componente è che ora l’istanza del ViewModel non viene più fornita dal passaggio per parametri: il valore delle proprietà esposte viene ora ottenuto utilizzando il metodo Bind.

Shared/ToDoFormComponent.razor

|  |
| --- |
| @inherits MvvmComponentBase<ToDoItemsListViewModel>  <h3>Nuovo ToDo</h3>  <EditForm Model="@Newtodoitem" OnValidSubmit="@CreaTodo">  <DataAnnotationsValidator />  <ValidationSummary />  <p>  <div class="form-field">  <label for="descrizione">Descrizione</label>  <InputText id="descrizione" @bind-Value=@Newtodoitem.Description />  </div>  </p>  <p>  <div class="form-field">  <label for="duedate">Due Date</label>  <InputDate id="duedate" @bind-Value=@Newtodoitem.DueDate ParsingErrorMessage="Inserire un data valida" />  </div>  </p>  <button type="submit">Salva</button>  </EditForm>  @code {  private ToDoItem Newtodoitem = new ToDoItem();  void CreaTodo()  {  BindingContext.AddToDoItem(Newtodoitem);  Newtodoitem = new ToDoItem();  }  } |

In questo componente l’unica modifica eseguita è relativa e relativa al metodo per richiamare il metodo AddToDoItem: in questo caso si è utilizzato l’oggetto BindingContext.

Vantaggi nell’adozione di BlazorMVVM

L'adozione della libreria ora mostrata permette di ottenere un codice più compatto e leggibile.

Infatti non risulta più essere necessario esplicitamente sottoscrivere all’evento PropertyChanged del ViewModel nella pagina principale index.razor per avere la visualizzazione aggiornata: la libreria si occuperà di questa funzionalità, nonché di propagare le notifica a tutti i componenti coinvolti.

Anche la classe ViewModel beneficia dall’adozione di BlazorMVVM: ora è possibile usare come tipo per la lista ToDoItemList la ObservableCollection, poiché ora l’evento CollectionChanged viene correttamente gestito per eseguire alla bisogna il refresh della tabella.

Grazie a questo non è più necessario emettere un evento PropertyChanged all’interno della proprietà e tantomeno dentro il metodo privato AddToDoItem.

Questo si traduce in un codice più semplice e più rispondente agli algoritmi implementativi.

Conclusioni

Blazor non è stato progettato per utilizzare in modo nativo il pattern MVVM, ma è ugualmente possibile il suo utilizzo.

Questa adozione porta a una serie di vantaggi, tra cui anche quello di portare le modalità di sviluppo tipiche del mondo desktop nell’ambito Web-SPA, in modo da riutilizzare competenze e logiche di funzionanto.

Il codice completo dell'esempio è disponibile su GitHub.

Linkografia

Suggestion - Make Blazor MVVM Friendly

<https://github.com/dotnet/aspnetcore/issues/16331>

Git MvvmBlazor

https://github.com/chris579/MvvmBlazor