10/04/20 - Versione 2 - WIP

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, pur se non viene fornito di default alcun tipo di supporto nativo alla sua adozione.

Questa mancanza probabilmente 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 avvantaggiare 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” (vedere linkografia).

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

Grazie a questo progetto vedremo come l’utilizzo del pattern MVVM in Blazor risulta essere facile e intuitivo, pur senza avere l’ausilio di alcun strumento nativo fornito dal framework, e permette di ottenere diversi vantaggi legati all’organizzazione del codice e delle logiche di funzionamento.

A completamento introdurremo un progetto di unit test che, grazie all’adozione del pattern MVVM, risulta favorito nella sua implementazione.

Descrizione del pattern

Prima di descrivere come implementare il pattern in un progetto Blazor scriveremo 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 i seguenti punti tra quelli principali per adottare il pattern MVVM nell’ambito dello sviluppo Blazor.

* Possibilità di utilizzare gli unit-test per verificare il comportamento del ViewModel
* Adozione di modalità di sviluppo tipiche del mondo desktop nell’ambito Web-SPA, in modo da riutilizzare competenze e logiche di funzionamento di alto livello.
* Riutilizzo di parte di codice già esistente con altri progetti dotati di user interface (Es.: WPF/Xamarin)

Condividere un ViewModel tra progetti Blazor e per esempio WPF/Xamarin risulta molto difficoltoso, poiché alcuni oggetti sono supportati in modo efficace in solo un ambito.

Per esempio, Blazor non supporta nativamente ICommand e ObservableCollection, mentre WPF/Xamarin non è in grado di comprendere l’oggetto Navigator.

Purtuttavia grazie all’adozione del pattern MVVM è possibile strutturare il codice in modo tale da permettere una condivisione anche parziale delle logiche di funzionamento, per esempio adottando pattern decorator in modo tale da creare uno strato per specializzare il ViewModel per la piattaforma. Questo permetterebbe di riutilizzare le logiche di funzionamento di alto livello del ViewModel.

Tra gli svantaggi principali che si possono apprezzare, invece, è che per progetti piccoli, o dotati di poche funzionalità, l’adozione del pattern 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 avvisato ogni qualvolta un valore tra quelli esposti dal ViewModel varia, in modo tale da poter eseguire le azioni necessarie per aggiornare la visualizzazione del controllo associato.

Per questo motivo all’interno della classe occorre prevedere un meccanismo in grado di far 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 al Blazor-component: come detto all’inizio, il framework non mette a disposizione alcun strumento particolare per favorire l’adozione di alcun pattern di programmazione, e tra questi MVVM.

Pertante per eseguire nel momento opportuno il refresh della visualizzazione è necessario sottoscrivere esplicitamente l’evento OnPropertyChanged.

Modifichiamo a tal scopo 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 si è sottoscritto l’evento PropertyChanged, associando come hanlder direttamente il 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 l’argomento dell’evento, 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 classe 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.

Introduzione agli Unit Test in Blazor

Lo scopo dei test è quello di verificare il corretto comportamento del codice secondo le specifiche, anche a fronte di modifiche successive che avvengono su di esso.

Detto in altri termini i test devono evidenziare non solo il corretto comportamento degli algoritmi implementati, ma anche che l’esito di questi non varino nel tempo, anche quando il codice viene modificato per implementare nuove funzionalità o sistemare bug di funzionamento.

Esemplificando parecchio è possibile identificare due tipologie di test.

* End-To-End
* Unit Test

Con il primo tipo si verifica il funzionamento globale del software interagendo esclusivamente con la user interface messa a disposizione per gli utilizzatori.

In altri termini è possibile affermare che usando questo tipo di verifica si segue esclusivamente la prospettiva dell’utilizzatore finale che inserisce i dati ed esegue azioni, verificando che il risultato ottenuto sia identico a quello atteso.

Si potrebbe pensare che questi tipo di test, soprattutto nell’ambito web, sia possibile solo manualmente, ma in realtà esistono framework appositi che simulano il comportamento umano: uno tra tutti è selenium (vedere linkografia).

L’altra tipologia di test utilizza, invece, un differente approccio: si occupa di verificare il corretto funzionamento di ogni modulo costitutivo del codice, come per esempio di ogni singola classe utilizzata o metodo di questa.

L’idea fondante è che quando è possibile stabilire il corretto funzionamento di ogni parte costitutiva allora è possibile avere la sicurezza che anche il comportamento globale dell’applicativo possa essere considerato affidabile.

Ogni tipologia di test ha i suoi pregi e difetti.

I test end-to-end sono in generale più lunghi da implementare ed eseguire, ma propongono anche implicitamente una verifica del funzionamento globale dei vari pezzi costitutivi del software connessi insieme: infatti possono incorrere problematiche relative all’integrazione dei vari moduli costitutivi anche a fronte del funzionamento corretto di questi se presi singolarmente.

Di contro gli unit-test, di contro, sono molto più veloci da eseguire, e la loro implementazione richiede una progettazione del codice complessivo che incrementa la “separation of concern” dei vari moduli.

Più difficile stabilire quali dei due tipi di test sia meglio: ovviamente la soluzione migliore è averli entrambi, poiché ognuno ha i suoi pregi, ma purtroppo poi nella pratica inevitabilmente se ne favorirà solo un tipo.

In Blazor sono possibili entrambi gli approcci, e nel seguito vedremo un esempio di implementazione di unit test per l’applicativo scritto: in questo modo sarà possibile verificare come l’adozione del pattern MVVM favorisca in modo sostanziale l’adozione di questo tipo di test.

Unit Test per ToDoListMVV

Alla solution creata in precedenza aggiungiamo un nuovo progetto preso dal template “MSTest Project .Net Core” che chiameremo UnitTestProjectBlazorToDoListMVVM: qui troveranno posto tutti gli unit test delegati a verificare il corretto funzionamento della classe ViewModel scritta in precedenza.

Un volta ottenuto questo progetto questo aggiungiamo a questo il riferimento al progetto BlazorToDoListMVVMLib creato in precedenza.

Infine modifichiamo la classe UntTest che è stata creata grazie al template selezionato prima.

|  |
| --- |
| [TestClass]  public class UnitTest1  {  [TestMethod]  public void ToDoItemsListViewModel\_NotifyPropertyChanged()  {  bool eventWasRaised = false;  var ToDoListViewModel = new ToDoItemsListViewModel();  Assert.IsInstanceOfType(ToDoListViewModel, typeof(INotifyPropertyChanged));  ToDoListViewModel.PropertyChanged += (sender, e) =>  {  if (e.PropertyName == nameof(ToDoItemsListViewModel.Title))  {  eventWasRaised = true;  }  };  ToDoListViewModel.Title = "Nuoto Titolo";    Assert.IsTrue(eventWasRaised);    }  [TestMethod]  public void ToDoItemsListViewModel\_AddToDoItem()  {  var ToDoListViewModel = new ToDoItemsListViewModel();  var NewToDoItem = new ToDoItem  {  Description = "Nuova attività"  };  ToDoListViewModel.AddToDoItem(NewToDoItem);  Assert.AreEqual(ToDoListViewModel.ToDoItemList.Count(), 1);  Assert.AreEqual (ToDoListViewModel.ToDoItemList.FirstOrDefault(), NewToDoItem);  }  } |

Nel primo metodo abbiamo scritto uno unit test che verifica che la classe ViewModel implementi nel modo corretto l’intefaccia InotifyPropertyChanged,

Nel metodo ToDoItemsListViewModel\_NotifyPropertyChanged per primo ci si assicura che la classe implementi l’interfaccia, successivamente si verifica che l’evento PropertyChanged previsto da questa sia correttamente emesso assegnando un valore alla proprietà Title.

Il secondo metodo ToDoItemsListViewModel\_AddToDoItem verifica, invece, il corretto funzionamento del metodo pubblico AddToDoItem, delegato a inserite nuovi ToDoItem nella lista dei todo.

Dopo averlo utilizzato ci si assicura che il numero degli item sia aumentato di un’unità e che quest’unica unita sia uguale all’istanza inserita della classe ToDoItem.

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, primo tra tutti il fatto di esemplificare la creazione di unit test da applicare al ViewModel.

Il codice completo dell'esempio è disponibile su GitHub.

Linkografia

Suggestion - Make Blazor MVVM Friendly

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

Selenium

<https://www.selenium.dev/>

Unit-test vs end-to-end test

https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html