C# MAUI (Multi-platform App UI) – Sviluppo Multipiattaforma

Slide 1: Introduzione a .NET MAUI

.NET Multi-platform App UI (MAUI) è un framework **multipiattaforma** di Microsoft per la creazione di applicazioni **native** per dispositivi mobili e desktop usando C# (con XAML opzionale) [1]. In pratica, .NET MAUI consente di sviluppare con un'unica base di codice app eseguibili su **Windows, Android, iOS, macOS** e persino su piattaforme come Samsung Tizen [1], eliminando la necessità di mantenere progetti separati per ogni sistema operativo.

Introdotto come evoluzione di Xamarin.Forms, .NET MAUI unifica lo sviluppo mobile e desktop nel **ecosistema .NET**. Il framework fornisce controlli dell'interfaccia utente **nativi per ogni piattaforma**, garantendo un'esperienza utente coerente e prestazioni ottimali su ogni dispositivo [2]. In altre parole, un'app MAUI "scrivi una volta" utilizza sotto il cofano i controlli UI specifici (Android, iOS, ecc.) offrendo un look & feel e performance paragonabili alle app sviluppate nativamente per ciascuna piattaforma.

```
// Esempio introduttivo: creare e visualizzare un semplice Label in MAUI
Label messaggio = new Label { Text = "Ciao da .NET MAUI!", FontSize = 24 };
// Aggiunge il controllo Label alla pagina corrente (assumendo contesto di
ContentPage)
this.Content = messaggio;
```

Slide 2: Perché usare .NET MAUI (Motivazioni)

Scrivi una volta, esegui ovunque: la motivazione principale per usare .NET MAUI è poter utilizzare un **unico codice** in C# per tutte le piattaforme, riducendo drasticamente i costi e i tempi di sviluppo [3]. Non è più necessario sviluppare separatamente in Swift/Objective-C per iOS, Kotlin/Java per Android e altre tecnologie per desktop: con MAUI condividi logica di business e interfaccia, concentrandoti su un solo linguaggio e framework. Questo approccio aumenta la produttività e facilita la manutenzione, poiché ogni modifica al codice si riflette in tutte le piattaforme supportate.

UI nativa e funzionalità avanzate: .NET MAUI utilizza controlli nativi, offrendo interfacce utente fluide e reattive su ogni dispositivo [2]. Inoltre, include strumenti moderni come **Hot Reload** (ricaricamento rapido) che permette di vedere immediatamente le modifiche al codice o all'interfaccia senza ricompilare l'app [4]. Supporta anche l'integrazione con tecnologie web tramite **Blazor Hybrid** e fornisce API **multipiattaforma** (per sensori, file, rete, ecc.), semplificando l'accesso alle funzionalità dei dispositivi [5]. Tutto ciò si traduce in uno sviluppo più veloce, meno errori dovuti a duplicazione di codice e applicazioni più consistenti su tutte le piattaforme.

```
// Unico codice per tutte le piattaforme: ottenere informazioni sulla
piattaforma corrente
DevicePlatform piattaforma = DeviceInfo.Current.Platform;
```

```
Console.WriteLine($"Piattaforma corrente: {piattaforma}");
// Ad esempio, stamperà Android, iOS, WinUI, macCatalyst, etc. a seconda del
dispositivo
```

Slide 3: Requisiti e Installazione dell'Ambiente di Sviluppo

Per sviluppare con .NET MAUI è necessario predisporre un ambiente adeguato. Su **Windows**, occorre installare **Visual Studio 2022** (versione 17.3 o successiva) con il workload ".NET Multi-platform App UI (MAUI)" [6]. Ciò include il SDK .NET 7/8 e tutti i componenti per Android, iOS, macOS e Windows. Assicurarsi di avere l'ultima versione del .NET SDK compatibile (ad esempio .NET 8 per usufruire delle ultime funzionalità). In alternativa, è possibile usare Visual Studio 2022 per Mac (su macOS) o configurare VS Code con estensioni MAUI, anche se l'esperienza più completa si ha con Visual Studio.

Piattaforme specifiche: per compilare ed eseguire app iOS su Windows è richiesto un **Mac** configurato come "Mac Agent" di build (Pair to Mac), poiché il toolchain di Apple (Xcode) è necessario per creare l'app iOS [7]. Per Android, Visual Studio include l'emulatore e SDK Android: abilitate la modalità sviluppatore e la virtualizzazione hardware per performance ottimali dell'emulatore. Su Windows, è possibile eseguire le app MAUI come applicazioni desktop native (basate su WinUI 3). In sintesi, preparare l'ambiente significa installare gli strumenti giusti e i SDK delle piattaforme target, assicurando che **Android SDK**, **OpenJDK** e eventuali certificati Apple (per iOS/macOS) siano pronti.

```
// Verifica rapida dell'ambiente .NET a runtime
Console.WriteLine($".NET version: {Environment.Version}");
// Esempio di controllo piattaforma in fase di esecuzione
if (OperatingSystem.IsWindows())
    Debug.WriteLine("App in esecuzione su Windows");
```

Slide 4: Creazione di un Nuovo Progetto .NET MAUI

Una volta configurato l'ambiente, possiamo creare il primo progetto MAUI. In Visual Studio, selezionare **Crea un nuovo progetto** e scegliere il template **"App .NET MAUI"**. Nella configurazione, assegnare un nome al progetto e selezionare il **framework .NET (7 o 8)** desiderato. VS genererà un progetto *multipiattaforma singolo*: nella Solution explorer vedrete un solo progetto con cartelle **Platforms** (contenenti codice e risorse specifiche per Android, iOS, Windows, macOS) e cartelle condivise per codice comune (ad es. **Views**, **ViewModels**, **Resources**). Questo *Single Project* contiene già una pagina di esempio (MainPage) con un semplice contatore "Hello, World".

Il codice di bootstrap del progetto si trova in **Program.cs** e **App.xaml.cs**. In Program.cs viene configurato il **MauiApp** (builder, servizi DI, ecc.), mentre in App.xaml.cs la classe **App** (derivata da `Application`) imposta la pagina iniziale. Ad esempio, la pagina main viene assegnata a `MainPage`. In un nuovo progetto, MainPage contiene un pulsante che incrementa un contatore al click. Possiamo esaminare questo evento per capire la struttura base di un'app MAUI.

```
// Esempio dal template: App.xaml.cs imposta la pagina principale dell'app
public partial class App : Application
{
    public App()
    {
```

```
        InitializeComponent();
        MainPage = new MainPage(); // Imposta la pagina iniziale dell'app
(MainPage.xaml)
    }
}
// MainPage.xaml ha definito l'interfaccia iniziale; MainPage.xaml.cs
gestisce gli eventi.
```

Slide 5: Layout – StackLayout (Impilamento di Elementi)

Per costruire l'interfaccia utente, .NET MAUI fornisce **layout** che definiscono come disporre controlli sullo schermo. Uno dei più semplici è **StackLayout**, che impila elementi in verticale o in orizzontale. Usando `StackLayout` possiamo mettere una serie di controlli uno dopo l'altro. Ad esempio, un StackLayout con orientamento verticale disporrà i child dall'alto al basso (come una colonna), mentre con orientamento orizzontale li disporrà da sinistra a destra (come una riga). Questo layout è utile per creare interfacce lineari, come form o liste verticali di elementi.

Caratteristiche principali di StackLayout: supporta margini e padding (spaziatura interna ed esterna), allineamento (ad es. `HorizontalOptions` / `VerticalOptions` per allineare i figli) e scrolling implicito se il contenuto supera lo spazio disponibile verticalmente. È un layout semplice ma potente per strutturare velocemente la UI. Nell'esempio seguente, creiamo in codice un StackLayout verticale e vi aggiungiamo alcuni controlli Label come figli.

```
// Crea uno StackLayout verticale e aggiunge etichette in ordine
StackLayout layout = new StackLayout {
    Orientation = StackOrientation.Vertical,  // Orientamento verticale
    Padding = 10  // Spaziatura interna di 10
};
layout.Children.Add(new Label { Text = "Elemento 1" });
layout.Children.Add(new Label { Text = "Elemento 2" });
layout.Children.Add(new Label { Text = "Elemento 3" });
// Questo impilerà "Elemento 1", "Elemento 2", "Elemento 3" verticalmente
nella pagina
this.Content = layout;
```

Slide 6: Layout – Grid (Layout a Griglia)

Un altro layout fondamentale è **Grid**, che consente di posizionare controlli in righe e colonne simili a una tabella. Con Grid possiamo ottenere interfacce più complesse e allineamenti bidimensionali. Si definiscono le **RowDefinitions** e **ColumnDefinitions** (specificando dimensioni fisse, automatiche o proporzionali con `*`). Ogni controllo figlio viene posizionato indicando la riga e colonna (proprietà `Grid.Row` e `Grid.Column` in XAML, oppure metodi in codice). La Grid è utile per creare layout responsivi: ad esempio, due colonne affiancate su schermi larghi che diventano una colonna singola su schermi stretti (adattando definizioni).

In MAUI, Grid funziona in modo simile a WPF/Xamarin.Forms. Possiamo usare span per far estendere un elemento su più colonne o righe (`Grid.ColumnSpan` / `RowSpan`). Nell'esempio seguente, creiamo

una griglia 2x2: due righe e due colonne, e posizioniamo quattro controlli Label nelle celle. Notare l'uso di `*` per righe/colonne a dimensione proporzionale (occuperanno lo spazio rimanente disponibile).

```
// Crea una griglia con 2 righe e 2 colonne
Grid grid = new Grid {
    RowDefinitions = {
        new RowDefinition { Height = new GridLength(1,
GridUnitType.Star) },    // Prima riga: altezza variabile (*)
        new RowDefinition { Height =
GridLength.Auto }                          // Seconda riga: altezza auto (in
base contenuto)
    },
    ColumnDefinitions = {
        new ColumnDefinition { Width = new GridLength(1,
GridUnitType.Star) },  // Prima colonna: larghezza *
        new ColumnDefinition { Width = new GridLength(2,
GridUnitType.Star) }   // Seconda colonna: larghezza 2*
    }
};
// Aggiunge 4 label nelle 4 celle (riga, colonna specificate)
grid.Add(new Label { Text = "Cella 0,0" }, 0, 0);
grid.Add(new Label { Text = "Cella 0,1" }, 0, 1);
grid.Add(new Label { Text = "Cella 1,0" }, 1, 0);
grid.Add(new Label { Text = "Cella 1,1" }, 1, 1);
this.Content = grid; // Imposta la Grid come contenuto della pagina corrente
```

Slide 7: Concetti Base – Data Binding

Data Binding è un concetto chiave in MAUI (ereditato da XAML): consente di collegare le proprietà dell'interfaccia utente ai dati (proprietà del modello o viewmodel) in modo dichiarativo. In pratica, quando il dato sorgente cambia, l'UI si aggiorna automaticamente, e viceversa (per controlli interattivi) senza dover scrivere codice di aggiornamento manuale. Questo promuove la separazione tra logica e presentazione. Un esempio semplice è associare il testo di una Label al contenuto di un Entry (campo di testo): man mano che l'utente digita nell'Entry, la Label può mostrare lo stesso testo in tempo reale.

In XAML si usa la sintassi `{Binding Proprietà}`, mentre in C# si può ottenere lo stesso effetto creando oggetti `Binding` e impostandoli sui controlli. Per funzionare pienamente in modalità bidirezionale, l'oggetto di dati deve notificare i cambiamenti (implementando `INotifyPropertyChanged` – argomento che vedremo con MVVM). Sotto, impostiamo un binding tra un Entry e un Label direttamente in codice: la proprietà `Text` della Label seguirà sempre il valore `Text` dell'Entry.

```
// Creazione di un Entry (input) e di una Label collegata al testo dell'Entry
Entry nomeInput = new Entry { Placeholder = "Inserisci il tuo nome" };
Label nomeLabel = new Label();
// Associa la proprietà Text della Label alla proprietà Text dell'Entry (data
binding)
nomeLabel.SetBinding(Label.TextProperty, new Binding("Text", source:
```

```
nomeInput));
// Aggiunge i controlli alla pagina (esempio con uno StackLayout)
Content = new StackLayout {
    Children = { nomeInput, nomeLabel }
};
// Ora, digitando nell'Entry, la Label mostrerà lo stesso testo in tempo
reale.
```

Slide 8: Concetti Base – Navigazione tra Pagine

Le applicazioni reali spesso hanno più schermate/pagine. .NET MAUI offre meccanismi di **navigazione** per spostarsi tra pagine, simili a quelli di Xamarin.Forms. Il modello di base è una **stack di navigazione**: si può **spingere** una nuova pagina in cima allo stack (visualizzandola) o **togliere** (pop) l'ultima pagina per tornare indietro. MAUI mette a disposizione la classe `NavigationPage` e, in modo più avanzato, la **Shell** (che semplifica la gestione di navigazione complessa, menu a schede, flyout menu, ecc.). In generale, una volta avvolta la MainPage in un NavigationPage, possiamo chiamare `Navigation.PushAsync()` per passare a una nuova pagina, e l'utente potrà tornare indietro con `PopAsync()` o tramite il tasto "Back" di sistema.

Il sistema di navigazione mantiene automaticamente lo **stato storico** delle pagine. Si possono anche passare dati alle pagine di destinazione (es. dettagli di un elemento) – vedremo un esempio più avanti. Se si utilizza la **Shell** di MAUI, la navigazione avviene tramite la registrazione di rotte e l'uso di `Shell.Current.GoToAsync(...)`, che permette di navigare anche in modo non gerarchico. Di seguito un esempio semplice di navigazione tradizionale: premendo un bottone, si istanzia e visualizza una nuova pagina `DettagliPage`.

```
// All'interno di una ContentPage già avvolta in un NavigationPage:
private async void ApriDettagli(object sender, EventArgs e)
{
    // Crea la pagina di destinazione e la spinge nello stack di navigazione
    await Navigation.PushAsync(new DettagliPage());
}
// In DettagliPage, per tornare indietro, si può usare:
await Navigation.PopAsync();  // rimuove la pagina corrente dallo stack,
tornando indietro
```

Slide 9: Concetti Base – Gestione Eventi UI

La **gestione degli eventi** è fondamentale per rendere l'app interattiva. In .NET MAUI (come in altri framework UI .NET) i controlli espongono eventi (ad es. un pulsante genera l'evento `Clicked` quando viene premuto). Possiamo sottoscriverci a questi eventi per eseguire codice in risposta alle azioni dell'utente. Ci sono due modi tipici: definire un **event handler** nominato (ad esempio, un metodo `OnButtonClicked` nel code-behind associato via XAML o codice) oppure usare **lambda expressions** direttamente in codice per gestire l'evento inline.

Gli eventi forniscono parametri utili (sender, e args) a seconda del tipo. Ad esempio, un campo di testo (Entry) ha un evento `Completed` che scatta alla fine dell'inserimento (quando l'utente preme Invio sulla tastiera). La gestione eventi in MAUI segue il pattern .NET standard – se si ha familiarità con

Windows Forms o WPF, è simile. Nell'esempio seguente, aggiungiamo un handler al **Click** di un pulsante per aggiornare il testo e loggare un messaggio quando viene cliccato.

```
Button btnSaluto = new Button { Text = "Saluta" };
// Aggancio dell'evento Clicked con una lambda expression
btnSaluto.Clicked += (sender, args) =>
{
    btnSaluto.Text = "Cliccato!";               // Cambia il testo del
bottone
    Console.WriteLine("Bottone premuto.");      // Logica aggiuntiva (es.
stampa in console di debug)
};
// Aggiungere btnSaluto alla pagina per poterlo visualizzare e interagire
```

Slide 10: Controlli Base – Label

Un **Label** è un controllo semplice per visualizzare testo statico o dinamico in UI. Viene spesso usato per titoli, etichette descrittive di campi, messaggi all'utente, ecc. In .NET MAUI, Label corrisponde a un controllo di testo nativo (UILabel su iOS, TextView su Android, ecc.), quindi eredita le capacità e limitazioni dei controlli di testo di ogni piattaforma. Puoi personalizzare molte proprietà: **Text** (il contenuto testuale), **TextColor** (colore del testo), **FontSize** (dimensione carattere), **FontAttributes** (stile come grassetto/corsivo), **HorizontalTextAlignment/VerticalTextAlignment** (allineamento del testo), e altro.

I Label non sono interattivi (non cliccabili di default) e servono solo per mostrare informazioni. Possono però aggiornarsi via data binding o codice. Se il testo è lungo, si possono configurare proprietà come **LineBreakMode** per gestire l'andata a capo o il troncamento. Nell'esempio seguente, creiamo un Label con testo di benvenuto, con dimensione del carattere aumentata e colore modificato, e lo aggiungiamo alla pagina.

```
// Creazione di un Label con alcune proprietà impostate
Label titolo = new Label {
    Text = "Benvenuto su .NET MAUI!",
    TextColor = Colors.Blue,       // Imposta il colore del testo (blu)
    FontSize = 32,                 // Font grande per titolo
    HorizontalTextAlignment = TextAlignment.Center  // Testo centrato
orizzontalmente
};
// Aggiunta del label alla pagina (ipotizziamo dentro un layout esistente)
layout.Children.Add(titolo);
```

Slide 11: Controlli Base – Button

Un **Button** rappresenta un pulsante cliccabile dall'utente. È uno dei controlli più usati per avviare azioni (salvataggio, invio, navigazione, ecc.). In MAUI, un Button è reso con i controlli nativi di ciascuna piattaforma (UIButton su iOS, Button su Android, ecc.), mantenendo così l'aspetto e i comportamenti standard. Le proprietà principali includono **Text** (la scritta sul pulsante), **TextColor**, **FontSize**, **BackgroundColor** (colore sfondo), e anche proprietà per arrotondare gli angoli (**CornerRadius**). Ma

l'aspetto cruciale è l'evento **Clicked**: possiamo associare un handler per eseguire codice quando l'utente preme il pulsante.

I pulsanti possono anche essere disabilitati (proprietà **IsEnabled = false**), se vogliamo impedirne temporaneamente l'uso. Inoltre, MAUI supporta i **Command** per i Button, integrandosi col pattern MVVM (lo vedremo nella sezione MVVM). Nell'esempio seguente, creiamo un Button con un testo e configuriamo l'evento Clicked per aggiornare il suo testo ad ogni pressione (implementando un semplice contatore di click).

```
// Creazione di un Button e gestione del click
int conteggio = 0;
Button btnConteggio = new Button { Text = "Premimi" };
btnConteggio.Clicked += (s, e) =>
{
    conteggio++;
    btnConteggio.Text = $"Premuto {conteggio} volte";
};
layout.Children.Add(btnConteggio);  // Aggiunge il pulsante ad un layout
esistente
// Ad ogni click, il testo del bottone viene aggiornato con il numero di
pressioni effettuate.
```

Slide 12: Controlli Base – Entry (Campo di Testo)

L'**Entry** è il controllo per l'input di testo singola riga (analogo a un `<input type="text">` HTML). Permette all'utente di inserire brevi testi, come nomi, email, password, ecc. Principali proprietà includono **Text** (il contenuto attuale), **Placeholder** (testo grigio mostrato quando il campo è vuoto, es: "Inserisci nome"), **IsPassword** (se true, maschera il testo con pallini, utile per password), **Keyboard** (per selezionare il tipo di tastiera virtuale appropriata, ad esempio numerica, email, telefono).

Entry espone eventi utili: **TextChanged** (mentre il testo cambia, ad ogni carattere) e **Completed** (quando l'utente ha finito di inserire e preme Invio o sposta il focus). Questi eventi aiutano a reagire all'input, ad esempio validando in tempo reale o aggiornando altri controlli via data binding. Nell'esempio qui sotto, configuriamo un Entry per email con placeholder e un evento Completed: quando l'utente termina di digitare l'email e conferma, eseguiamo un controllo di validità semplificato e stampiamo il risultato.

```
Entry emailEntry = new Entry {
    Placeholder = "Inserisci la tua email",
    Keyboard = Keyboard.Email  // Mostra tastiera ottimizzata per email su
mobile
};
emailEntry.Completed += (s, e) =>
{
    string email = emailEntry.Text;
    bool valida = email.Contains("@");
    Console.WriteLine(valida ? $"Email valida: {email}" : "Email non
valida");
```

```
};
layout.Children.Add(emailEntry);
```

Slide 13: Controlli Base – ListView e Liste di Elementi

Per mostrare elenchi di dati scorrevoli, .NET MAUI fornisce controlli come **ListView** (ereditato da Xamarin.Forms) e il più moderno **CollectionView**. Qui consideriamo **ListView**, che visualizza una lista verticale di elementi e gestisce in automatico lo scrolling. ListView è utile per presentare collezioni di dati omogenei, ad esempio una lista di contatti, di prodotti, ecc. Può utilizzare modelli di cella (template) per definire la visualizzazione di ogni elemento. In modalità semplice, se si lega l'ItemsSource a una lista di stringhe, ogni stringa verrà mostrata come testo.

Proprietà principali: **ItemsSource** (la fonte dati, ad esempio una `IEnumerable`), **SelectedItem** (elemento attualmente selezionato), e eventi come **ItemSelected** (quando l'utente tocca un elemento). ListView supporta anche l'**aggiornamento pull-to-refresh**, raggruppamento, e altro. Tuttavia, in MAUI si preferisce spesso CollectionView per prestazioni e flessibilità migliorate. Nell'esempio, creiamo una lista di stringhe e la assegnamo a un ListView; ogni voce verrà visualizzata come una riga di testo.

```
// Creiamo una lista di stringhe come esempio di dati
List<string> colori = new List<string> { "Rosso", "Verde", "Blu", "Giallo" };
// Inizializziamo un ListView e assegniamo la sorgente dati
ListView listView = new ListView {
    ItemsSource = colori
};
// (Facoltativo) gestiamo l'evento di selezione di un elemento
listView.ItemSelected += (s, e) =>
{
    if (e.SelectedItem != null)
        Console.WriteLine($"Selezionato: {e.SelectedItem}");
};
this.Content = listView;  // Mostriamo il ListView come contenuto della
pagina
```

Slide 14: Altri Controlli Comuni (Slider, Switch, ecc.)

Oltre ai controlli base, .NET MAUI include molti altri controlli utili. Ad esempio, **Slider** permette di selezionare un valore continuo spostando un cursore (utile per volume, luminosità), **Switch** è un interruttore on/off (per impostazioni booleane), **Picker** consente di scegliere un valore da una lista a comparsa, **DatePicker/TimePicker** per selezionare date e orari, **CheckBox** per opzioni selezionabili, **Stepper** per incrementare/decrementare valori numerici con frecce, e **Editor** per testi multilinea. Questi controlli seguono lo stesso principio: proprietà per stato e valore corrente, ed eventi per reagire all'interazione. Ad esempio, Switch ha la proprietà **IsToggled** e l'evento **Toggled**.

Tutti i controlli supportano data binding, facilitando l'aggiornamento bidirezionale tra UI e dati. È importante conoscere l'esistenza di questi controlli per costruire interfacce più ricche. Nell'esempio sottostante, creiamo uno Switch (interruttore) per attivare/disattivare una funzionalità e agganciamo l'evento Toggled per eseguire un'azione quando cambia stato.

```
// Creazione di uno Switch (on/off) e gestione del suo evento
Switch toggle = new Switch { IsToggled = false };
toggle.Toggled += (sender, e) =>
{
    bool acceso = e.Value;  // e.Value indica il nuovo stato (true = On,
false = Off)
    Console.WriteLine(acceso ? "Funzionalità attivata" : "Funzionalità
disattivata");
    // Qui possiamo abilitare/disabilitare funzionalità nell'app in base allo
stato
};
// Aggiungere lo Switch ad un layout per renderlo visibile
```

Slide 15: Pattern MVVM – Introduzione

Nello sviluppo di applicazioni con MAUI è fortemente consigliato adottare il pattern **MVVM (Model-View-ViewModel)**. MVVM aiuta a separare la logica di business e di presentazione dall'interfaccia utente, rendendo il codice più organizzato, testabile e manutenibile. In MVVM: - **Model**: rappresenta i dati e le entità dell'applicazione (es. classi che modellano un prodotto, un utente, etc.), spesso recuperati da database o servizi. - **View**: è la UI, cioè le pagine XAML/C# con controlli che visualizzano i dati. - **ViewModel**: funge da intermediario tra View e Model; contiene la logica dell'interfaccia, i comandi (azioni da eseguire) e i dati da mostrare, esponendo proprietà alle quali la View può fare data binding.

La ViewModel non conosce la View (non fa riferimenti diretti a controlli), e la View non contiene logica se non la dichiarazione di binding. Questo decoupling permette ad esempio di testare le ViewModel in isolamento. In MAUI possiamo creare una ViewModel come semplice classe C# con proprietà e implementare l'interfaccia `INotifyPropertyChanged` per notificare i cambiamenti (così la UI si aggiorna). Vediamo una semplicissima ViewModel di esempio, che rappresenta uno studente con due proprietà (Nome e Matricola).

```
// Esempio di ViewModel (modello di vista) semplice
public class StudenteViewModel
{
    public string Nome { get; set; }        // nome dello studente
    public int Matricola { get; set; }      // numero di matricola
    // In un MVVM completo, implementeremo INotifyPropertyChanged per
notificare cambiamenti
    // e potremmo aggiungere metodi o comandi per operazioni (es:
SalvaStudenteCommand).
}
```

Slide 16: MVVM – Implementazione di INotifyPropertyChanged

Per aggiornare automaticamente la UI quando i dati cambiano, le proprietà esposte dalla ViewModel devono notificare i cambiamenti. Ciò si ottiene implementando l'interfaccia **INotifyPropertyChanged**, che definisce l'evento `PropertyChanged`. Ogni volta che una proprietà viene modificata, la ViewModel "alza" questo evento, e i controlli in binding su quella proprietà si aggiornano. Questo evita di dover

aggiornare manualmente il controllo. Spesso si crea una base class ViewModel che implementa INotifyPropertyChanged, da cui far derivare le proprie VM.

Nell'esempio seguente, definiamo una ViewModel **ContatoreViewModel** che mantiene un contatore intero. Quando il contatore cambia (proprietà `Valore`), eseguiamo `OnPropertyChanged("Valore")` per notificare la UI. Notare il pattern: la proprietà set modifica il campo e chiama OnPropertyChanged col nome della proprietà. `OnPropertyChanged` a sua volta invoca l'evento `PropertyChanged` se qualcuno è iscritto (tipicamente il binding della UI). Questo è il meccanismo standard per data binding notificabile.

```
using System.ComponentModel;
public class ContatoreViewModel : INotifyPropertyChanged
{
    private int _valore;
    public int Valore {
        get => _valore;
        set {
            if(_valore != value) {
                _valore = value;
                OnPropertyChanged(nameof(Valore));  // Notifica la UI che
"Valore" è cambiato
            }
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string nomeProprieta)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(nomeProprieta));
    }
}
```

Slide 17: MVVM – Associare ViewModel alla View (BindingContext)

Per far sì che i binding definiti nella View (pagina XAML o codice UI) puntino alle proprietà della ViewModel, bisogna **collegare la ViewModel alla View**. In MAUI questo si fa assegnando un'istanza della ViewModel alla proprietà **BindingContext** della pagina (o di un contenitore). Ad esempio, nella classe code-behind di una ContentPage, possiamo creare la ViewModel e fare `this.BindingContext = miaViewModel;`. Da quel momento, la pagina (e i suoi controlli) utilizzerà quella ViewModel come sorgente dati per i binding.

In alternativa, in XAML possiamo impostare la ViewModel usando markup o servizi di dependency injection, ma è importante capire il concetto: BindingContext è il "contesto" che i `{Binding ...}` nella UI guardano per trovare le proprietà. Una volta impostato, se la ViewModel implementa INotifyPropertyChanged (come visto), qualsiasi aggiornamento alle sue proprietà notificherà la UI. Nell'esempio sottostante, creiamo una semplice ViewModel che espone un messaggio e la colleghiamo a una Label in pagina via binding.

```
// Definizione di una semplice ViewModel con una proprietà
public class MessaggioViewModel : INotifyPropertyChanged {
    public string Messaggio { get; set; } = "Messaggio iniziale";
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string nome) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nome));
}
// Nella pagina (code-behind), impostiamo il BindingContext:
this.BindingContext = new MessaggioViewModel();
// In XAML o in code, un Label con Text="{Binding Messaggio}" mostrerà
"Messaggio iniziale"
// Se in futuro MessaggioViewModel.Messaggio cambia e chiama
OnPropertyChanged, il Label si aggiorna.
```

Slide 18: MVVM – Command e Azioni senza Code-Behind

Nel pattern MVVM, idealmente evitiamo di gestire eventi UI direttamente nel code-behind della View. Invece, utilizziamo i **Command** esposti dalla ViewModel per reagire alle azioni dell'utente. Un Command in MAUI è tipicamente un oggetto che implementa `ICommand` (ad esempio la classe `Command` integrata). Si può associare un Command a un controllo UI (es: Button) tramite la proprietà `Command` (in XAML `{Binding NomeCommand}`): quando l'utente clicca il bottone, il framework invocherà il comando legato, che eseguirà la logica definita nella ViewModel.

Questo favorisce testabilità e separazione (la logica del click risiede nella VM, non nella code-behind della pagina). I Command possono anche abilitare/disabilitare pulsanti tramite `CanExecute`. Nell'esempio seguente, aggiungiamo alla ViewModel del contatore un comando **IncrementaCommand** che quando eseguito incrementa il contatore. In XAML, un Button potrebbe poi fare `Command="{Binding IncrementaCommand}"` invece di utilizzare l'evento Clicked.

```
using System.Windows.Input;
public class ContatoreViewModel : INotifyPropertyChanged
{
    public int Valore { get; private set; }
    public ICommand IncrementaCommand { get; }
    public ContatoreViewModel() {
        // Inizializza il comando associandolo a un'azione
        IncrementaCommand = new Command(() => {
            Valore++;
            OnPropertyChanged(nameof(Valore)); // Notifica aggiornamento
        });
    }
    // ... (INotifyPropertyChanged implementato come visto prima)
}
```

Commenti: in questo esempio, `Command(() => {...})` è una scorciatoia fornita da MAUI per creare ICommand. Ogni volta che il comando viene eseguito (dalla UI), incrementiamo `Valore` e notifichiamo la UI. Il Button collegato mostrerà quindi il nuovo valore se è in binding a `Valore`.

Slide 19: Navigazione in Pratica – Shell

.NET MAUI introduce la **Shell** come modo ottimizzato per strutturare e navigare tra le pagine dell'app. Shell fornisce un contenitore che può definire l'intera navigazione dell'app tramite un file XAML (AppShell.xaml) o in codice. Consente di creare facilmente menu a pannello (flyout menu), tab bar (schede in basso) e gestire rotte di navigazione con URL semantici. Utilizzando Shell, possiamo registrare percorsi (Route) per le pagine e poi navigare chiamando `Shell.Current.GoToAsync("nomeRoute")` da qualsiasi punto, senza dover passare oggetti NavigationPage esplicitamente.

Con Shell, la **MainPage** dell'app è tipicamente impostata a un `AppShell` che definisce `<FlyoutItem>` o `<TabBar>` con le varie sezioni/pagine. La Shell gestisce automaticamente la visualizzazione di menu e la creazione della Navigation stack. Un vantaggio notevole è la semplicità nel passare parametri tramite query URI. Per usare Shell, registriamo le rotte delle pagine secondarie in AppShell (in code-behind o XAML). Nell'esempio in codice seguente, registriamo una route e poi usiamo Shell per navigare.

```
// Nel costruttore di AppShell (code-behind), possiamo registrare rotte:
Routing.RegisterRoute(nameof(DettagliPage), typeof(DettagliPage));
// Da una ViewModel o dove abbiamo accesso a Shell.Current, navigare alla
pagina:
await Shell.Current.GoToAsync(nameof(DettagliPage));
// In alternativa, con parametri: Shell.Current.GoToAsync("DettagliPage?
Id=5");
```

Slide 20: Navigazione in Pratica – NavigationPage

Prima di Shell, il metodo classico per abilitare la navigazione era utilizzare **NavigationPage**. In .NET MAUI è ancora disponibile e utile per casi semplici o per mantenere uno stack di pagine senza implementare Shell. Per utilizzarlo, si avvolge la pagina principale dell'app in un NavigationPage, ad esempio: `MainPage = new NavigationPage(new MainPage())`. Ciò fornisce automaticamente una **barra di navigazione** in alto con titolo e pulsante "back" quando necessario. Da quel punto, all'interno delle pagine è possibile usare `Navigation.PushAsync(new AltraPage())` per spostarsi avanti e `Navigation.PopAsync()` per tornare indietro.

NavigationPage gestisce la pila di pagine in modo implicito. È utile per workflow lineari (es. una pagina di elenco -> pagina di dettaglio). Può anche essere combinato con Shell in alcune circostanze, ma in generale si sceglie uno dei due approcci. Sotto vediamo come impostare la MainPage con NavigationPage e poi come chiamare la navigazione in avanti.

```
// In App.xaml.cs, per abilitare la navigazione tradizionale:
public App()
{
    InitializeComponent();
    MainPage = new NavigationPage(new MainPage());  // Avvolge MainPage in
NavigationPage
}
// Esempio di navigazione in avanti da MainPage (code-behind):
```

```
private async void VaiDettagli(object sender, EventArgs e)
{
    await Navigation.PushAsync(new DettagliPage());  // Apre DettagliPage
sopra MainPage
}
```

Slide 21: Navigazione – Passare Dati tra Pagine

Spesso durante la navigazione è necessario **trasferire dei dati** alla pagina di destinazione (es: l'ID di un elemento da mostrare nei dettagli). Ci sono diversi modi per farlo in MAUI. Uno semplice è tramite il **costruttore** della pagina: si può definire il costruttore di DettagliPage con parametri e passare i valori al momento di creazione. Un altro metodo, se si usa Shell, è tramite **query parameters** nella stringa di navigazione (configurando la pagina di destinazione per ricevere quei parametri con `[QueryProperty]`). Inoltre, è possibile utilizzare un servizio condiviso o un ViewModel comune (per scenari master-detail).

Nell'approccio più diretto, supponiamo di voler passare un oggetto o un identificativo. Possiamo modificare `DettagliPage` aggiungendo un costruttore personalizzato. Nell'esempio, la pagina dettaglio accetta un ID e magari caricherà i dati corrispondenti. Mostriamo come chiamare PushAsync passando un parametro via costruttore.

```
// Pagina di destinazione (DettagliPage) con costruttore personalizzato
public class DettagliPage : ContentPage
{
    private int _prodottoId;
    public DettagliPage(int prodottoId) {
        _prodottoId = prodottoId;
        InitializeComponent();
        // Usare _prodottoId per caricare dettagli, ad esempio da database o
servizio
    }
}
// Navigazione dalla pagina origine, passando un ID al costruttore:
int idSelezionato = 42;
await Navigation.PushAsync(new DettagliPage(idSelezionato));
```

Slide 22: Dati Locali – Gestione File

Le applicazioni spesso necessitano di salvare dati localmente sul dispositivo, ad esempio file di configurazione, cache o documenti utente. .NET MAUI offre accesso al file system del dispositivo in modo sicuro: tramite la classe `FileSystem` possiamo ottenere percorsi appropriati, come **AppDataDirectory**, una cartella dedicata all'app. Una volta ottenuto un percorso, possiamo usare le normali API di .NET (System.IO) per leggere/scrivere file. Ad esempio, `File.WriteAllText` per scrivere rapidamente testo, `FileStream` per operazioni binarie, ecc.

Occorre ricordare che su piattaforme mobile come Android e iOS ci sono sandbox: l'app ha accesso solo alle proprie cartelle (salvo permessi speciali). AppDataDirectory punta automaticamente a una cartella che persiste tra esecuzioni ma è privata all'app. Nell'esempio seguente, salviamo un piccolo testo in un

file locale e poi lo rileggiamo, stampando il contenuto. Questo potrebbe rappresentare, ad esempio, il salvataggio di note o configurazioni.

```
using Microsoft.Maui.Storage;
string percorsoCartella =
FileSystem.Current.AppDataDirectory;                // cartella dati dell'app
string percorsoFile = Path.Combine(percorsoCartella, "note.txt");
// Scrive del testo su un file nella cartella locale
File.WriteAllText(percorsoFile, "Promemoria: studiare MAUI ogni giorno");
// Legge il contenuto del file
string contenuto = File.ReadAllText(percorsoFile);
Console.WriteLine("Contenuto file: " + contenuto);
```

Slide 23: Dati Locali – Preferenze e Impostazioni

Per salvare piccole quantità di dati (come impostazioni utente, preferenze, flag booleani) è consigliabile usare le **Preferences** fornite da MAUI (ereditate da Xamarin.Essentials). Le Preferences permettono di memorizzare coppie chiave-valore in modo semplice e persistente (tipicamente nei SharedPreferences su Android, NSUserDefaults su iOS, registry/file su Windows). Sono ideali per salvare ad esempio l'ultimo utente loggato, preferenze dell'app (tema scuro/chiaro), ecc.

L'uso è immediato: si chiama `Preferences.Default.Set("chiave", valore)` per salvare e `Preferences.Default.Get<T>("chiave", valoreDefault)` per leggere. Supportano tipi base (int, string, bool, double, ecc.). I dati salvati persistono tra diverse esecuzioni dell'app. Nell'esempio sotto, salviamo un nome utente nelle preferenze e lo recuperiamo all'avvio successivo.

```
using Microsoft.Maui.Storage;
// Salvataggio di una preferenza (es: nome utente)
Preferences.Default.Set("username", "MarioRossi");
// ... in un momento successivo (anche dopo riavvio app):
string utente = Preferences.Default.Get("username", "sconosciuto");
Console.WriteLine("Utente salvato: " + utente);
// Risulterà "MarioRossi" se in precedenza settato, altrimenti
"sconosciuto" (default).
```

Slide 24: Dati Remoti – Chiamate HTTP (REST API)

Le applicazioni moderne spesso comunicano con servizi remoti (ad es. API REST, servizi web, cloud). In .NET MAUI possiamo utilizzare le classi .NET standard come **HttpClient** per effettuare richieste HTTP. Ad esempio, per ottenere dati JSON da un server REST (es. un elenco di elementi) e poi mostrarli nell'app. HttpClient consente di inviare GET, POST, PUT, DELETE, ecc. e ricevere risposte. È importante usare le chiamate in modo **asincrono** (`await`) per non bloccare il thread UI durante l'attesa della risposta. Inoltre, dovremo parsare i dati ricevuti (spesso in formato JSON) in oggetti C# utilizzando magari `System.Text.Json` o librerie come Newtonsoft.Json.

Nell'esempio seguente, effettuiamo una richiesta GET ad un endpoint remoto che restituisce una stringa JSON. Poi utilizziamo `JsonSerializer` per deserializzare questa stringa in una lista di oggetti

di un certo tipo (qui ipotizziamo una classe Articolo). Naturalmente, in un caso reale, la classe Articolo dovrebbe corrispondere al formato dei dati JSON.

```
using System.Net.Http;
using System.Text.Json;
HttpClient client = new HttpClient();
string url = "https://api.example.com/articoli";  // URL dell'API da chiamare
try {
    string json = await client.GetStringAsync(url);
    // Deserializza il JSON in una lista di oggetti Articolo (definiti
altrove)
    List<Articolo> articoli =
JsonSerializer.Deserialize<List<Articolo>>(json);
    Console.WriteLine($"Scaricati {articoli.Count} articoli dal server.");
} catch(Exception ex) {
    Console.WriteLine("Errore nella chiamata HTTP: " + ex.Message);
}
```

Slide 25: Dati Remoti – Aggiornare l'UI con Dati Esterni

Dopo aver ottenuto dati da un servizio remoto, il passo successivo è **aggiornare l'interfaccia utente** per mostrarli. In MVVM, si avrà tipicamente un **ViewModel** con una proprietà (es: ObservableCollection) che contiene la lista di elementi da visualizzare in una ListView/CollectionView. Quando i dati arrivano dall'API, l'app li deserializza e li assegna a questa proprietà, e grazie al binding la UI si aggiorna. È importante ricordare di eseguire gli aggiornamenti UI sul **thread principale**. In C#, dopo un `await` su HttpClient, il contesto UI di solito ritorna automaticamente (grazie a SynchronizationContext su piattaforme UI) – quindi possiamo aggiornare direttamente le proprietà bindate. Se si utilizza un Task senza await, servirà `MainThread.BeginInvokeOnMainThread`.

Nell'esempio, immaginiamo una ViewModel con un'ObservableCollection di Articolo. Implementiamo un metodo asincrono `CaricaDatiAsync` che scarica i dati e li aggiunge alla collezione osservabile. ObservableCollection notificherà automaticamente la ListView bindata. Questo illustra come collegare il risultato della chiamata remota all'aggiornamento dell'interfaccia.

```
public class ArticoliViewModel : INotifyPropertyChanged
{
    public ObservableCollection<Articolo> ElencoArticoli { get; set; } =
new();
    public async Task CaricaDatiAsync() {
        // Scarica dati (chiamata HTTP simulata)
        string json = await httpClient.GetStringAsync("https://
api.example.com/articoli");
        var nuovi = JsonSerializer.Deserialize<List<Articolo>>(json);
        // Aggiorna collezione sul thread UI
        MainThread.BeginInvokeOnMainThread(() => {
            ElencoArticoli.Clear();
            foreach(var art in nuovi)
                ElencoArticoli.Add(art);
```

```
        });
    }
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Slide 26: Funzionalità Native – Accesso alla Fotocamera e Immagini

.NET MAUI integra **APIs native** (originariamente Xamarin.Essentials) per accedere all'hardware del dispositivo in modo multipiattaforma. Una funzionalità tipica è l'accesso alla **fotocamera** o alla libreria immagini. Tramite l'interfaccia `MediaPicker` possiamo permettere all'utente di scattare una foto o sceglierne una esistente. Ad esempio, `MediaPicker.Default.PickPhotoAsync()` apre la galleria immagini, mentre `CapturePhotoAsync()` avvia la fotocamera. Queste funzioni restituiscono un oggetto `FileResult` che rappresenta il file dell'immagine. Possiamo poi aprire uno stream del file e caricarlo in un controllo Image nell'app.

Ricordiamo di aggiungere nel manifest i permessi necessari (fotocamera, accesso storage) a seconda della piattaforma. L'esempio sottostante mostra come scattare una foto con la fotocamera e ottenere uno stream per visualizzarla nell'app.

```
using Microsoft.Maui.Media;
FileResult foto = await
MediaPicker.Default.CapturePhotoAsync();  // Avvia la fotocamera
if (foto != null)
{
    // Ottiene uno stream del file fotografico
    using Stream stream = await foto.OpenReadAsync();
    // Crea un ImageSource dall'immagine catturata e la assegna a un
controllo Image
    ImageSource imgSource = ImageSource.FromStream(() => stream);
    myImageControl.Source = imgSource;
    // 'myImageControl' è un oggetto Image definito nella UI per mostrare la
foto
}
```

Slide 27: Funzionalità Native – Geolocalizzazione (GPS)

Molte app richiedono servizi di **geolocalizzazione**, ad esempio per ottenere la posizione GPS dell'utente. MAUI fornisce l'API `Geolocation` per recuperare la posizione corrente. Il metodo principale è `Geolocation.Default.GetLocationAsync()`, che restituisce un oggetto `Location` contenente latitudine, longitudine, altitudine e altri dati. La prima volta, potrebbe chiedere permessi all'utente (che vanno anche dichiarati nel manifest Android/iOS). È possibile specificare la precisione richiesta (grossolana vs fine) e un timeout. Inoltre, vi è `GetLastKnownLocationAsync` per ottenere una posizione in cache velocemente.

Nell'esempio seguente, recuperiamo la posizione attuale e la usiamo. Bisogna gestire i possibili eccezioni: ad esempio, permesso negato, GPS non attivo, device senza sensore, ecc. Nel nostro semplice caso, stampiamo latitudine e longitudine se disponibili.

```
using Microsoft.Maui.Devices.Sensors;
try {
    Location posizione = await Geolocation.Default.GetLocationAsync();
    if (posizione != null) {
        double lat = posizione.Latitude;
        double lon = posizione.Longitude;
        Console.WriteLine($"Posizione attuale - Lat: {lat}, Lon: {lon}");
    } else {
        Console.WriteLine("Impossibile ottenere la posizione.");
    }
}
catch (Exception ex) {
    Console.WriteLine($"Errore GPS: {ex.Message}");
}
```

Slide 28: Funzionalità Native – Altre (Vibrazione, Torcia, Telefono...)

Oltre a fotocamera e GPS, .NET MAUI (via Essentials) espone molte altre funzionalità native dei dispositivi con interfaccia unificata: - **Vibrazione**: possiamo far vibrare il telefono con `Vibration.Default.Vibrate()`, utile per feedback tattili. - **Flash/Torcia**: con `Flashlight.Default.TurnOnAsync()` e `TurnOffAsync()` si controlla la torcia (se disponibile). - **Sensori**: Accelerometro, Giroscopio, Magnetometro, ecc., accessibili tramite eventi (`Accelerometer.ReadingChanged`...). - **Telefono/Comunicazione**: ad esempio, `PhoneDialer.Default.Open(number)` per aprire il dialer telefonico con un numero, `Sms.Default.ComposeAsync` per inviare SMS precompilati, `Email.Default.ComposeAsync` per email. - **Browser**: aprire link esterni con `Launcher.Default.OpenAsync(uri)`.

Queste API semplificano l'uso dell'hardware senza dover scrivere codice specifico per Android/iOS. L'esempio seguente mostra come far vibrare il dispositivo per una breve durata, ad esempio come feedback di un'azione.

```
using Microsoft.Maui.ApplicationModel;
using Microsoft.Maui.Devices;
// Fa vibrare il dispositivo per 500 millisecondi
Vibration.Default.Vibrate(TimeSpan.FromMilliseconds(500));
// Suggerimento: è buona pratica arrestare la vibrazione eventualmente:
Vibration.Default.Cancel();
```

Slide 29: Gestione Piattaforme – Progetto Unico e Multi-Target

Uno dei punti di forza di MAUI è la **single-project** architecture: un unico progetto .NET contiene tutto il codice, gestendo internamente i target multipli (Android, iOS, Windows, macOS). Nel file .csproj vengono specificati i target frameworks (ad es. net7.0-android, net7.0-ios, net7.0-windows10.0.19041.0, net7.0-maccatalyst). In base alla piattaforma di compilazione, il SDK include automaticamente i file appropriati dalla cartella **Platforms**. Ciò semplifica la gestione, avendo un solo posto dove definire risorse e codice condiviso e riducendo duplicazioni [8].

Tuttavia, a volte è necessario scrivere codice differenziato per piattaforma (ad esempio chiamare API specifiche non coperte da MAUI). Per questo, .NET offre le direttive di compilazione condizionale **#if**. Possiamo racchiudere porzioni di codice con `#if ANDROID`, `#if IOS`, `#if WINDOWS`, ecc., che verranno incluse solo quando si compila per quella piattaforma. In alternativa, si possono creare file separati (parti di classi `partial`) nelle cartelle Platforms, mantenendo il codice specifico isolato. L'esempio seguente illustra l'uso di direttive per eseguire codice dipendente dalla piattaforma.

```
string messaggio = "Esecuzione su: ";
#if ANDROID
messaggio += "Android";
#elif IOS
messaggio += "iOS";
#elif MACCATALYST
messaggio += "macOS";
#elif WINDOWS
messaggio += "Windows";
#else
messaggio += "piattaforma sconosciuta";
#endif
Console.WriteLine(messaggio);
// Questo codice aggiunge il nome della piattaforma corrente al messaggio e
lo stampa
```

Slide 30: Gestione Piattaforme – Codice Specifico e Dependency Injection

Se una funzionalità nativa non è fornita direttamente dalle API multipiattaforma di MAUI, possiamo implementarla usando codice specifico per piattaforma e poi fornire un'interfaccia comune all'app. Un approccio è utilizzare **Dependency Injection** con interfacce. Si definisce un'interfaccia (es: `INotificatore`) nel progetto condiviso e implementazioni per ciascuna piattaforma (es: `AndroidNotificatore` in Platforms/Android, `IosNotificatore` in Platforms/iOS) che usino le API native (NotificationManager su Android, UIKit su iOS, ecc.). Durante la configurazione (in `MauiProgram.CreateMauiApp`), registriamo l'implementazione specifica in base alla piattaforma con `builder.Services`.

Così facendo, nel codice condiviso possiamo chiedere il servizio `INotificatore` e ottenerne automaticamente l'istanza corretta per la piattaforma corrente. Questo favorisce un codice pulito e un unico punto di accesso. Nell'esempio seguente, vediamo una registrazione condizionale di un servizio e come consumarlo.

```
// Registrazione di un servizio specifico per piattaforma in MauiProgram.cs
#if ANDROID
builder.Services.AddSingleton<INotificatore, AndroidNotificatore>();
#elif IOS
builder.Services.AddSingleton<INotificatore, IosNotificatore>();
#endif

// Utilizzo nella ViewModel (grazie a DI, supponendo costruttore con
```

```
INotificatore)
public class NotificaViewModel {
    private readonly INotificatore _notificatore;
    public NotificaViewModel(INotificatore notificatore) {
        _notificatore = notificatore;
    }
    public void InviaNotifica(string msg) {
        _notificatore.MostraNotifica(msg);  // Chiama implementazione nativa
corretta
    }
}
```

Slide 31: Debugging e Hot Reload

Durante lo sviluppo con MAUI, gli strumenti di **debug** sono essenziali per trovare e correggere errori. Usando Visual Studio, possiamo eseguire l'app in modalità Debug su emulatori o dispositivi reali, impostare **breakpoint** nel codice C# e ispezionare variabili, stacktrace, ecc. Questo funziona analogamente ad altri progetti .NET: l'app si interrompe al breakpoint e possiamo analizzare lo stato. MAUI aggiunge la comodità del **XAML Hot Reload** e **.NET Hot Reload**: queste funzionalità permettono di modificare l'interfaccia XAML o anche il codice C# di definizione UI durante l'esecuzione, e vedere le modifiche applicate immediatamente [5]. Ad esempio, cambiare un colore in XAML riflette subito sull'app in esecuzione, riducendo il ciclo modifica-compila-esegui.

Un altro strumento utile è la **Output window** e il logging: possiamo usare `Debug.WriteLine` o `Console.WriteLine` (in modalità debug) per stampare messaggi diagnostici che appaiono nel log. Su device Android/iOS collegati potremmo anche usare strumenti platform-specific (adb logcat, etc.), ma Visual Studio semplifica mostrando l'output. In caso di crash, Visual Studio interromperà l'esecuzione e spesso fornirà lo stacktrace dell'eccezione non gestita. Sotto, un esempio di utilizzo di un codice eseguito solo in debug e di un messaggio di log.

```
int x = 5;
int y = 0;
#if DEBUG
Console.WriteLine("Modalità Debug attiva");  // Questo codice viene incluso
solo in Debug
#endif

try {
    int z = x / y;
} catch(Exception ex) {
    Debug.WriteLine($"Errore rilevato: {ex.Message}");
    // In debugger possiamo ispezionare 'ex' per maggiori dettagli
}
```

Slide 32: Testing dell'App (Unit Test)

Scrivere test automatizzati è una buona pratica per assicurare la qualità del codice. Con MAUI, possiamo creare progetti di **Unit Test** separati (librerie .NET tradizionali) per testare la logica nelle ViewModel, nei

servizi e in qualunque classe non legata direttamente all'interfaccia. Ad esempio, se abbiamo una classe Calcolatrice con un metodo Somma, possiamo verificarne il funzionamento con un test. Framework come **NUnit** o **xUnit** possono essere usati. Visual Studio può eseguire questi test e integrarsi nel flusso CI/CD. Le ViewModel sono particolarmente adatte ai test: essendo separate dalla UI e usando binding, possiamo simulare cambi di proprietà o esecuzione di command e verificare i risultati (ad es. che un certo valore sia aggiornato).

Si possono anche fare test strumentali (UITest) con strumenti come Xamarin.UITest o Appium, che pilotano l'app installata per simulare interazioni reali, anche se sono più complessi da impostare. Nella fase iniziale, puntiamo sui unit test della logica. L'esempio mostra un test usando *NUnit* per verificare un metodo di somma.

```
[Test]
public void Somma_DueNumeri_RestituisceSommaCorretta()
{
    // Arrange
    int a = 2, b = 3;
    // Act
    int risultato = Calcolatrice.Somma(a, b);
    // Assert
    Assert.AreEqual(5, risultato, "La somma di 2 e 3 dovrebbe essere 5");
}
```

Slide 33: Distribuzione – Preparare l'App per il Deploy

Una volta completato lo sviluppo e i test, arriva il momento di **distribuire l'app** agli utenti. .NET MAUI consente di compilare pacchetti di distribuzione per ciascuna piattaforma: - **Android:** si genera un file .apk o **Android App Bundle (.aab)** per la pubblicazione. L'APK è adatto per installazioni dirette su dispositivi, mentre l'AAB è richiesto per pubblicare sul Play Store [9]. Bisogna compilare in **Release**, firmare l'app con una keystore (creare un certificato se non esiste) e abilitare l'**IL trimming** se si vuole ridurre la dimensione. - **iOS:** si produce un pacchetto **.ipa** (simile a un archivio zip di app) per TestFlight o App Store. Serve iscriversi al Apple Developer Program, creare un profilo di provisioning e un certificato di firma. La pubblicazione su App Store include passare per Xcode Application Loader o tooling automatico [10]. MAUI su Windows può creare l'IPA appoggiandosi a un Mac. - **Windows:** l'app può essere pubblicata come eseguibile/autonomo in una cartella, oppure pacchettizzata in **MSIX** per la distribuzione (ad es. Microsoft Store) [11]. MSIX offre benefici come aggiornamenti differenziali e installazione semplificata.

Durante il deploy, ricordare di aggiornare le informazioni di versione dell'app (versione e build) nel Manifest/csproj. Nell'esempio seguente, vediamo come recuperare nome e versione dell'app (impostati nelle proprietà) tramite `AppInfo` – utile per visualizzarli o verificarli prima di pubblicare.

```
using Microsoft.Maui.ApplicationModel;
string nomeApp = AppInfo.Current.Name;
string versioneApp = AppInfo.Current.VersionString;
Console.WriteLine($"Preparazione distribuzione: {nomeApp} v{versioneApp}");
// Esempio: "Preparazione distribuzione: MiaApp v1.0"
```

Slide 34: Pubblicazione su Google Play (Android)

Per distribuire l'app Android al grande pubblico, si utilizza il **Google Play Store**. Il processo prevede di generare un **Android App Bundle (AAB)** in modalità Release, firmato con una chiave privata. In Visual Studio, configurare la build Release per Android, assicurarsi di avere specificato nell'.csproj l'**ApplicationId** (il package name univoco, es. com.nome.app) e incrementato VersionCode/ VersionName. Dopo la build, si ottiene un file .aab (o .apk se richiesto). Questo file va caricato sulla console Google Play, creando un'app, compilando i dettagli (descrizione, immagini, rating content, ecc.). Google Play eseguirà la verifica e, una volta approvata, l'app sarà pubblicata per gli utenti.

È importante **firmare digitalmente** l'app: se fatto tramite Visual Studio, bisogna creare un Keystore (.keystore) e configurare la firma in progetto (o usare la firma Play Console). Inoltre, testate la versione Release su dispositivi reali per assicurarsi che trimming e linker non abbiano rimosso codice necessario. Una volta pubblicata, gestite le versioni aggiornando VersionCode ad ogni rilascio. Esempio: impostare i valori di versione nell'app.

```
// Esempio (in csproj o Manifest Android, non codice runtime) - solo per
illustrare:
<?xml version="1.0" encoding="utf-8" ?>
<manifest package="com.azienda.miaapp" android:versionCode="2"
android:versionName="1.1" ...>
  <!-- versionCode 2 indica update (deve aumentare), versionName 1.1 è la
versione visibile -->
</manifest>
// Nota: Questo snippet XML va nel AndroidManifest.xml generato in fase di
publish,
// qui mostrato come concetto di impostazione delle versioni per Google Play.
```

Slide 35: Pubblicazione su Apple App Store (iOS)

Distribuire un'app iOS/macOS richiede di passare per l'ecosistema Apple. Per iOS, occorre un **account sviluppatore Apple** (a pagamento annuale). Si configura un **App ID** unico e un **Profilo di provisioning** che lega l'app al certificato di firma e ai dispositivi (per beta testing) [10]. In Visual Studio (collegato a un Mac), si compila in **Release per iOS**, generando un archivio (**.ipa** file). Questo pacchetto contiene la app firmata e pronta per la distribuzione. Si può testare via TestFlight caricandolo su App Store Connect e invitando utenti beta. Per la pubblicazione sull'App Store, tramite App Store Connect si inseriscono tutte le info (icona, descrizione, categorie, privacy) e si sottopone l'app per revisione Apple.

Processi chiave: assicurarsi di incrementare la **versione build** (CFBundleVersion) e versione marketing (CFBundleShortVersionString) nell'Info.plist/csproj. Gestire eventuali **capabilities** (come permessi di background, iCloud, ecc.) nel progetto. Apple può impiegare tempo per la review, e richiede che l'app rispetti linee guida stringenti. Nell'esempio, vediamo come definire nella configurazione dell'app la versione e build per iOS.

```
// Esempio (snippet di Info.plist o csproj per iOS) – indicativo, non codice
C# reale
<Placemark>
    <key>CFBundleShortVersionString</key> <string>1.1</string>   <!--
```

```
versione visibile -->
    <key>CFBundleVersion</key> <string>2</string>                <!-- build
number -->
</Placemark>
// Questi valori vanno aggiornati per ogni release destinata all'App Store.
```

Slide 36: Distribuzione su Windows e macOS (Desktop)

Per le piattaforme desktop, la distribuzione può avvenire sia tramite store ufficiali sia tramite installazione diretta. Su **Windows**, .NET MAUI consente di pubblicare l'app in formato **MSIX**, che è il pacchetto di app moderno supportato dallo Store Microsoft e dall'installazione offline [11]. L'MSIX include l'app e le dipendenze, con vantaggi come aggiornamenti automatici e installazione pulita. In Visual Studio, si può usare il progetto di packaging o il comando `dotnet publish -f net7.0-windows10.0.19041.0 -c Release` e poi utilizzare strumenti come MSIX Packaging Tool. In alternativa, è possibile pubblicare come una cartella self-contained (che contiene un .exe e librerie) da distribuire via installer custom.

Su **macOS**, le app MAUI girano come Mac Catalyst apps. La distribuzione può avvenire creando un **.app** bundle o un **.pkg** installer [12]. Per pubblicare sul Mac App Store, serve un account sviluppatore Apple (macOS) e firmare il .app con certificato Apple, similmente a iOS. Per distribuzione diretta (fuori dallo store), conviene fornire un .dmg o .pkg firmato e notarizzato da Apple (richiesto su macOS moderne per superare Gatekeeper). In entrambi i casi, testare su macOS reale è fondamentale.

```
// (Pseudo codice per illustrare rilevazione piattaforma desktop)
if (DeviceInfo.Current.Platform == DevicePlatform.WinUI)
{
    Console.WriteLine("Preparazione pacchetto MSIX per Windows...");
}
if (DeviceInfo.Current.Platform == DevicePlatform.MacCatalyst)
{
    Console.WriteLine("Generazione .app/.pkg per macOS...");
}
```

Slide 37: Risorse del Progetto – Immagini e File Incorporati

Le **risorse** (immagini, icone, font, file di configurazione) in MAUI si gestiscono attraverso la cartella **Resources** del progetto unico. Ad esempio, in **Resources/Images** possiamo aggiungere immagini (.png, .jpg, .svg): il build system di MAUI le prenderà e genererà automaticamente le versioni ottimali per ogni piattaforma e diverse densità (per Android). Queste immagini diventano accessibili tramite il loro nome file (senza estensione) nel codice XAML o C#. Allo stesso modo, i **font personalizzati** inseriti in **Resources/Fonts** vengono registrati (nel MauiProgram.cs con `ConfigureFonts`) e possono essere usati assegnando la famiglia definita.

Per file come suoni, file json o altri asset, si può usare **Resources/Raw** (che li include tali e quali) oppure **Embedded resources** classiche .NET. Gestire le risorse centralmente permette di mantenere coerenza visiva e supportare temi. Nell'esempio seguente, carichiamo un'immagine denominata *logo.png* aggiunta in Resources/Images semplicemente impostando la proprietà Source di un controllo Image.

```
// Uso di un'immagine incorporata nelle risorse (Resources/Images/logo.png)
Image logoImage = new Image { Source = "logo.png" };
// In MAUI non serve specificare percorsi per immagini in Resources/Images,
// basta usare il nome file. Il framework seleziona l'immagine giusta per la
piattaforma/dimensione.
layout.Children.Add(logoImage);
```

Slide 38: Stili e Temi Grafici

Per mantenere un aspetto coerente nell'app e supportare modalità **chiaro/scuro**, .NET MAUI consente di definire **stili** e **risorse globali**. Possiamo usare il ResourceDictionary dell'app (in App.xaml) o dei singoli controlli per definire colori, dimensioni e stili riutilizzabili. Ad esempio, definire un colore primario una volta e riutilizzarlo ovunque tramite StaticResource o DynamicResource. Possiamo anche definire Style per controlli (es: uno style per tutte le Label dei titoli con certe proprietà di font).

MAUI supporta inoltre la reazione al tema del sistema: con `App.Current.UserAppTheme` si può forzare un tema, oppure usare `DynamicResource` e `AppThemeBinding` per fornire valori diversi in light/dark mode (es. testo nero su sfondo chiaro, testo bianco su scuro). Qui sotto, un esempio programmatico: impostiamo un colore nelle risorse applicative e lo utilizziamo per colorare il testo di un Label. Inoltre, usiamo `SetAppThemeColor` per definire un colore che cambia a seconda del tema.

```
// Definizione di una risorsa colore globale
Application.Current.Resources["ColorePrimario"] = Colors.Teal;
// Uso della risorsa in un controllo Label
Label titolo = new Label { Text = "Titolo" };
titolo.TextColor = (Color)Application.Current.Resources["ColorePrimario"];

// Impostazione di un colore tema-specifico (scuro/chiaro) su un controllo
Label testo = new Label { Text = "Testo adattivo" };
text.SetAppThemeColor(Label.TextColorProperty, Colors.Black, Colors.White);
// In modalità chiara il testo sarà nero, in modalità scura sarà bianco
automaticamente.
```

Slide 39: Ciclo di Vita delle Pagine

Le pagine in .NET MAUI (derivate da `ContentPage`) hanno un loro **ciclo di vita** che è utile conoscere per inizializzare dati o rilasciare risorse. I metodi principali (override) forniti sono: - `OnAppearing()`: chiamato quando la pagina sta per apparire sullo schermo (diventa visibile). Qui possiamo aggiornare l'interfaccia, caricare dati freschi, avviare animazioni. - `OnDisappearing()`: chiamato quando la pagina sta per scomparire (l'utente naviga altrove). Qui possiamo salvare lo stato, fermare timers, annullare richieste in corso per non consumare risorse. Questi metodi permettono di reagire all'entrata/uscita della pagina dallo stack di navigazione. Ad esempio, in OnAppearing di una pagina di elenco potremmo ricaricare i dati dal database per mostrare sempre info aggiornate quando l'utente arriva.

Nel codice seguente, mostriamo come override di OnAppearing in una pagina possa essere usato per loggare o aggiornare qualcosa quando la pagina diventa attiva.

```
public class ListaPage : ContentPage
{
    protected override void OnAppearing()
    {
        base.OnAppearing();
        Console.WriteLine("ListaPage è ora visibile - aggiornamento dati.");

viewModel.CaricaElementi();  // Esempio: ricarica la lista ogni volta che si
mostra
    }
    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        Console.WriteLine("ListaPage sta per chiudersi.");
        // Esempio: potremmo salvare uno stato temporaneo qui
    }
}
```

Slide 40: Ciclo di Vita dell'Applicazione

Oltre al ciclo di vita delle singole pagine, c'è il ciclo di vita globale dell'applicazione (l'intero processo). Su mobile, l'app può passare attraverso stati: **avvio**, **in background**, **ripresa**, **chiusura**. In Xamarin.Forms esistevano i metodi `OnStart`, `OnSleep`, `OnResume` nella classe App. In .NET MAUI, questi possono essere gestiti tramite eventi di `Application` o piattaforma (ad esempio su Android Activity events). Microsoft.Maui.Essentials fornisce l'evento statico `Application.Current.Suspending/Resuming` per alcuni scenari. L'idea è di salvare dati quando l'app sta andando in background (per evitare perdita di stato se il sistema la termina) e ripristinare/storicizzare informazioni al resume.

Ad esempio, un'app potrebbe salvare l'ora di ultimo utilizzo per poi, al riavvio, calcolare se mostrare schermate di login o aggiornare dati. O ancora, interrompere servizi di localizzazione o musica quando l'app non è in primo piano. Nell'esempio, ipotizziamo di implementare in App.xaml.cs un override di un evento di sospensione per salvare una preferenza.

```
public partial class App : Application
{
    public App() {
        InitializeComponent();
        // ... inizializzazione normale
    }

    protected override void OnSleep()
    {
        // Chiamato quando l'app sta andando in background (su alcune
piattaforme)
        Preferences.Default.Set("ultimo_accesso", DateTime.Now);
        base.OnSleep();
    }

    protected override void OnResume()
```

```
    {
        // Chiamato quando l'app ritorna in primo piano
        DateTime ultimo = Preferences.Default.Get("ultimo_accesso",
DateTime.MinValue);
        Console.WriteLine($"Ripresa app. Ultimo uso: {ultimo}");
        base.OnResume();
    }
}
```

Slide 41: Strumenti e Librerie Aggiuntive (Community Toolkit)

L'ecosistema .NET MAUI è arricchito da librerie open-source che semplificano lo sviluppo. Una di queste è il **.NET MAUI Community Toolkit**, una raccolta di estensioni, controlli e helper utili. Ad esempio, offre controlli aggiuntivi (CameraView, Expander, etc.), convertitori e comportamenti (per evitare di scrivere codice banale in VM), e il famoso **Markup Extensions C#** per definire UI fluentemente in codice anziché XAML. Inoltre, fornisce attributi come `[ObservableProperty]` e `[RelayCommand]` tramite CommunityToolkit.Mvvm, che permettono di ridurre il boilerplate nelle ViewModel (generando automaticamente proprietà con INotifyPropertyChanged e comandi).

Oltre al toolkit, esistono librerie di **terze parti** come Syncfusion, Telerik, DevExpress che offrono controlli avanzati (grafici, datagrid, scheduler) compatibili con MAUI. Si possono includere via NuGet. Saper integrare queste librerie amplia le capacità dell'app senza dover reinventare la ruota. Nell'esempio seguente, vediamo come grazie al **MVVM Toolkit** possiamo definire una ViewModel molto concisa usando attributi: il codice generato dietro le quinte implementerà INotifyPropertyChanged e i command per noi.

```
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Input;
public partial class UtenteViewModel : ObservableObject
{
    [ObservableProperty]
    private string nome;   // Genera automaticamente la proprietà Nome con
notifica

    public UtenteViewModel() {
        nome = "Anonimo";
    }

    [RelayCommand]
    private void Saluta() {
        Console.WriteLine($"Ciao, {Nome}!");  // Command eseguibile dalla
View (collegato a un Button)
    }
}
// Grazie a [ObservableProperty] e [RelayCommand], non scriviamo manualmente
OnPropertyChanged o ICommand
```

Slide 42: Accessibilità e Localizzazione

Un'app professionale deve considerare l'**accessibilità**, cioè essere usabile anche da persone con disabilità (es. non vedenti, ipovedenti, motorie). .NET MAUI consente di aggiungere **Automation Properties** ai controlli, come il nome accessibile, suggerimenti, ecc. Questi vengono letti dai **screen reader** (come TalkBack su Android o VoiceOver su iOS). Ad esempio, possiamo impostare `AutomationProperties.SetName(button, "Invia messaggio")` per fornire una descrizione significativa di un pulsante, se la sua etichetta non è sufficiente. Inoltre, usare font scalabili e contrasto adeguato aiuta. MAUI supporta anche la preferenza di sistema "dimensioni testo maggiori" automaticamente se usiamo font size in NamedSize.

La **localizzazione** permette di tradurre l'app in più lingue. Si può usare il classico approccio .resx (resource file) con CultureInfo oppure librerie come Multilingual Toolkit. In MAUI, le risorse .resx funzionano e possiamo ottenere stringhe localizzate e bindarle. Bisogna impostare il thread UI sulla cultura giusta (`CultureInfo.CurrentUICulture`). Ad esempio, avere Resources.it.resx, Resources.en.resx e usare `Resource.App.Title` per ottenere il titolo nella lingua corrente. Sotto, un esempio di impostazione di un nome accessibile e di caricamento di una stringa localizzata (supponendo esistenza di risorse).

```
using Microsoft.Maui.Accessibility;
Button invioBtn = new Button { Text = "Invia" };
AutomationProperties.SetName(invioBtn, "Invia messaggio");
// Questo aiuta lo screen reader a fornire più contesto ("Invia messaggio"
invece di solo "Invia").

string titolo = AppResources.TitoloApp;  // Carica stringa localizzata dal
file di risorse (.resx) in base alla cultura attuale
this.Title = titolo;
```

Slide 43: Ottimizzazione delle Prestazioni

Per garantire un'esperienza utente fluida, bisogna considerare le **prestazioni** dell'app. Alcuni consigli: - **Evitare operazioni pesanti sul thread UI**: spostare calcoli intensi o accessi a rete/dati su thread in background (Task.Run, async/await) per non bloccare l'interfaccia. - **Usare collezioni ottimizzate**: preferire CollectionView a ListView per elenchi lunghi, perché supporta la virtualizzazione (carica solo gli elementi visibili). - **Ridurre l'overdraw grafico**: limitare layout annidati molto complessi, usare pochi livelli di Grid/Stack se possibile. - **Cache**: sfruttare caching di immagini (il controllo Image di MAUI ha CacheEnabled di default) e risultati di rete quando opportuno. - **Profili di performance**: MAUI consente di abilitare il profiler Mono o .NET e usare strumenti come Visual Studio Profiler o dotnet-counters per individuare colli di bottiglia.

Inoltre, per ridurre la dimensione dell'app, MAUI usa il **linker** (trimmer) in Release per rimuovere codice inutilizzato [13] . Bisogna testare bene in Release per evitare che il trimming tagli via codice necessario (in tal caso, usare attributi o file di configurazione linker). Sotto, un esempio che illustra l'esecuzione di un'operazione pesante off-thread e misurare il tempo impiegato.

```
using System.Diagnostics;
Stopwatch sw = new Stopwatch();
```

```
sw.Start();
// Esegui operazione pesante senza bloccare UI
int risultato = await Task.Run(() => CalcoloIntensivo());
sw.Stop();
Console.WriteLine($"Calcolo intensivo completato in {sw.ElapsedMilliseconds}
ms. Risultato = {risultato}");
labelRisultato.Text = $"Risultato: {risultato}";  // Aggiorna UI sul thread
principale
```

Slide 44: Gestione degli Errori e Logging

Robustezza significa anche gestire gli **errori** in modo appropriato. In un'app MAUI, possiamo incappare in eccezioni dovute a rete assente, file non trovati, permessi mancanti, ecc. È buona pratica utilizzare blocchi **try-catch** attorno alle operazioni a rischio (es. chiamate Http, accesso file, operazioni su sensori) per intercettare eccezioni ed evitarne la propagazione che porterebbe al crash dell'app. Possiamo poi informare l'utente con messaggi user-friendly (es. "connessione assente, riprova"). MAUI/Essentials fornisce anche alcune eccezioni specializzate (es. `FeatureNotSupportedException`).

Per debug e monitoraggio sul campo, l'uso di un sistema di **logging** aiuta. In .NET, possiamo utilizzare `Debug.WriteLine` in debug, oppure integrare librerie come Serilog, Microsoft.Extensions.Logging (MAUI supporta la Dependency Injection di un ILogger che possiamo utilizzare nelle VM). In produzione, potremmo inviare log o crash report a servizi esterni (AppCenter, Sentry, etc.). Nell'esempio seguente, gestiamo un possibile errore di rete in una chiamata API e logghiamo l'eccezione.

```
try {
    string response = await httpClient.GetStringAsync("https://
api.miosito.com/data");
    ElaboraRisposta(response);
}
catch (HttpRequestException httpEx) {
    Debug.WriteLine("Errore di rete: " + httpEx.Message);
    await Application.Current.MainPage.DisplayAlert("Connessione assente",
        "Non è stato possibile recuperare i dati. Verifica la connessione
internet.", "OK");
}
catch (Exception ex) {
    Debug.WriteLine("Eccezione non prevista: " + ex);
    // Gestione generica dell'errore (eventualmente mostrare messaggio
generico)
}
```

Slide 45: Adattare l'UI a Diversi Dispositivi (Responsive Design)

Con MAUI, l'applicazione può girare su smartphone, tablet e desktop con dimensioni di schermo molto varie. È importante progettare un'UI **responsiva**. Alcuni consigli: utilizzare layout adattivi (Grid con colonne che compaiono/scompaiono), controllare l'**idioma del dispositivo** (DeviceIdiom) per magari cambiare la disposizione degli elementi (ad esempio su Tablet mostrare un pannello affiancato invece di navigazione a pagina intera). MAUI fornisce `DeviceInfo.Current.Idiom` che può essere Phone,

Tablet, Desktop, TV, etc. Possiamo anche reagire all'orientamento dello schermo (verticale/orizzontale) usando la proprietà `DisplayInformation.Current.Orientation` da Microsoft.Maui.Devices.

Un'altra tecnica è usare margini e layout flessibili (FlexLayout o Grid con *). Inoltre, si può definire in XAML i* VisualStateManager* per stati differenti (ad es. stato Narrow vs Wide). L'esempio seguente illustra una decisione in codice: se l'app sta su telefono, si naviga a pagina dettaglio; se su tablet/desktop, si potrebbe mostrare il dettaglio affiancato.

```
using Microsoft.Maui.Devices;
if (DeviceInfo.Current.Idiom == DeviceIdiom.Phone) {
    await Shell.Current.GoToAsync(nameof(DetailPage));  // Su phone:
navigazione a pagina intera
} else if (DeviceInfo.Current.Idiom == DeviceIdiom.Tablet ||
DeviceInfo.Current.Idiom == DeviceIdiom.Desktop) {
    // Su schermi grandi: mostrare dettaglio in split view, ad esempio
affiancando due pannelli
    MostraDettaglioAffiancato(selectedItem);
}
```

Slide 46: Asincronismo e Gestione dei Thread

Le app devono rimanere reattive: per questo **asincronismo** e gestione thread sono cruciali. In MAUI, tutte le operazioni lunghe (rete, I/O, elaborazioni pesanti) dovrebbero usare `async/await` in modo da liberare il thread dell'interfaccia durante l'attesa. Abbiamo già accennato all'uso di `Task.Run` per spostare calcoli sul thread threadpool. Inoltre, per aggiornare l'UI da un thread secondario bisogna usare il **thread principale** (UI thread). MAUI fornisce `MainThread.BeginInvokeOnMainThread` per eseguire un'azione sul thread UI, qualora ci trovassimo in un contesto diverso. In molti casi, però, se si `await` un task, la continuazione torna sul thread originale (grazie al SynchronizationContext) – quindi aggiornare la UI subito dopo un await è lecito.

Gestire correttamente i token di cancellazione (`CancellationToken`) è utile per abortire operazioni asincrone (es. annullare una richiesta se l'utente naviga via). Nell'esempio, effettuiamo un'operazione simulata in background e aggiorniamo un controllo al termine, assicurandoci di farlo sul thread giusto.

```
// Simula un calcolo in background e aggiorna UI al termine
ProgressBar barra = new ProgressBar { Progress = 0.0 };
await Task.Run(() => {
    // Operazione lunga
    for(int i=0; i<=100; i++) {
        Thread.Sleep(20); // simulazione lavoro
        // Aggiornamento progress bar man mano (sul thread UI)
        MainThread.BeginInvokeOnMainThread(() => {
            barra.Progress = i / 100.0;
        });
    }
});
// Dopo il loop, siamo ancora dentro Task.Run. Il progress finale è gestito
sopra.
```

```
Console.WriteLine("Operazione asincrona completata");
// La ProgressBar viene aggiornata gradualmente senza congelare
l'interfaccia.
```

Slide 47: Esempio Pratico – Mini App “To-Do”

Mettiamo insieme i concetti appresi con un piccolo esempio di un’app To-Do (lista di cose da fare). Potremmo avere una pagina con un elenco di attività (ListView) e una casella di input + bottone per aggiungerne di nuove. La ViewModel associata (ToDoViewModel) avrà un’ObservableCollection di elementi ToDo e un comando per aggiungere elementi. Ogni elemento ToDo può essere un modello con proprietà Descrizione e Fatto (booleano). Navigazione: una pagina di dettaglio per modificare l’elemento, ecc. Pur senza implementare tutto, mentalmente percorriamo: definizione del Model, definizione della ViewModel con proprietà e comandi, binding nella View XAML ai controlli (ItemsSource per ListView, Text di Entry associato a proprietà nuovaVoce, Command del Button associato a AggiungiCommand).

Ad esempio, la ViewModel può avere: - `NuovaVoce` (string) bindata all’Entry. - `ElencoVoci` (ObservableCollection<ToDo>) bindata alla ListView. - `AggiungiVoceCommand` (ICommand) collegato al Button. Quando eseguito, aggiunge un nuovo ToDo a ElencoVoci e svuota `NuovaVoce`.

Ecco uno snippet semplificato di come potrebbe apparire la ViewModel e l’utilizzo di questi componenti:

```
public class ToDoItem { public string Testo { get; set; } public bool
Completato { get; set; } }

public class ToDoViewModel : INotifyPropertyChanged
{
    public ObservableCollection<ToDoItem> ElencoVoci { get; } = new();
    public string NuovaVoce { get; set; }
    public ICommand AggiungiVoceCommand { get; }
    public ToDoViewModel()
    {
        AggiungiVoceCommand = new Command(() =>
        {
            if (!string.IsNullOrWhiteSpace(NuovaVoce)) {
                ElencoVoci.Add(new ToDoItem { Testo = NuovaVoce, Completato
= false });
                NuovaVoce = string.Empty;
                OnPropertyChanged(nameof(NuovaVoce));
            }
        });
    }
    public event PropertyChangedEventHandler PropertyChanged;
    void OnPropertyChanged(string name) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

Slide 48: Risorse Utili e Link di Approfondimento

Terminata la lezione, è importante sapere dove trovare ulteriori **risorse** per approfondire MAUI: - **Documentazione Ufficiale Microsoft:** è il punto di riferimento primario. Il sito Microsoft Learn contiene guide passo passo, tutorial e reference API per .NET MAUI [14]. Ad esempio, la sezione *Build your first app* e le doc sui controlli specifici. - **Esempi di Codice:** Microsoft fornisce su GitHub una serie di sample ufficiali [15] su vari aspetti (Shell, CollectionView, ecc.). Analizzarli aiuta molto. - **Community e Forum:** la community MAUI è attiva su StackOverflow, Reddit (r/dotnetMAUI), e forum Microsoft Q&A. Qui si trovano soluzioni a problemi comuni e best practice condivise dagli sviluppatori. - **Blog e Video:** canali come dotnet YouTube, conferenze (es. .NET Conf focus on MAUI) e blog di esperti contengono articoli aggiornati, suggerimenti per scenari specifici e le novità delle versioni future. - **Toolkit e Librerie:** esplorare la documentazione del .NET MAUI Community Toolkit per scoprire i componenti aggiuntivi disponibili. Anche i provider di controlli (Syncfusion, etc.) spesso offrono documenti e esempi gratuiti per iniziare con i loro strumenti.

In aggiunta, mantenere .NET MAUI aggiornato (applicare gli update di versione e service release) è fondamentale, poiché migliorie e bugfix vengono rilasciati frequentemente.

```
// È possibile aprire link utili direttamente dall'app (ad esempio un
pulsante "Guida"):
await Launcher.Default.OpenAsync("https://learn.microsoft.com/dotnet/maui");
// Questo comando aprirebbe il browser sull'URL della documentazione
ufficiale .NET MAUI.
```

Slide 49: Best Practices e Consigli Finali

Per concludere, riepiloghiamo alcuni **best practice** nello sviluppo MAUI: - **Organizzazione del Codice:** mantieni il progetto pulito separando le View (pagine) dalle ViewModel e dai Model. Segui convenzioni di naming (es: NomePagina.xaml e NomePaginaViewModel.cs). - **Uso appropriato di async:** non bloccare mai il thread UI; usa async/await ovunque possibile per chiamate a servizi, I/O e lunghe elaborazioni. Considera l'utilizzo di `ConfigureAwait(false)` nelle librerie se necessario. - **Gestione Memoria:** fai Dispose o utilizza `using` per oggetti pesanti (es. Stream, DbConnection) per evitare memory leak, specialmente su mobile dove le risorse sono limitate. Ad esempio, immagini scaricate: liberare lo stream dopo l'uso. - **UI/UX consistente:** sfrutta stili e risorse globali per colori e font, in modo da poter cambiare facilmente temi o adattare il design. Testa l'app con dimensioni di font accessibilità maggiorate e in dark mode. - **Errori e Telemetria:** prevedi dove possono accadere errori (network, permessi negati) e gestiscili con feedback all'utente. Valuta l'integrazione di un servizio di crash reporting per ricevere info su eventuali eccezioni in produzione. - **Performance:** carica in modo pigro (lazy load) i dati quando possibile, e usa indicatori di attività (ActivityIndicator) per far capire all'utente che l'app non è bloccata ma sta lavorando. Questo migliora la percezione di velocità.

Ricorda che lo **sviluppo cross-platform** unifica molto, ma a volte è necessario testare specificatamente su ogni piattaforma per rifinire dettagli (ad esempio differenze di padding, comportamento tasti fisici come la back button Android, ecc.).

```
// Best practice esempio: usare 'using' per gestire risorse correttamente
using FileStream fs = File.OpenRead("file.dat");
// ... utilizza fs per leggere i dati ...
```

```
// Al termine del blocco 'using', il FileStream viene automaticamente chiuso
e rilasciato
```

Slide 50: Conclusione (Fine Presentazione)

Complimenti, hai completato questa panoramica approfondita su **.NET MAUI** e lo sviluppo multipiattaforma in C#! Abbiamo coperto le motivazioni per adottare MAUI, come preparare l'ambiente di sviluppo e creare un progetto, i concetti fondamentali di layout, controlli e navigazione, l'architettura MVVM per applicazioni scalabili, come accedere a dati locali e remoti, e integrare funzionalità native dei dispositivi. Abbiamo anche discusso come gestire le differenze tra piattaforme, come effettuare debug e test efficaci, e infine come distribuire la tua applicazione su vari store e dispositivi.

Questo è solo l'inizio: il mondo dello sviluppo cross-platform è in continua evoluzione. Ti incoraggiamo a sperimentare creando piccole app demo per fissare i concetti, consultare la documentazione ufficiale e partecipare alla community per apprendere trucchi ed esperienze altrui. Con MAUI, le tue competenze C# ti permetteranno di costruire app per qualsiasi piattaforma con un'unica codebase – un potente vantaggio nel mondo dello sviluppo moderno. **Buon codice e buona fortuna con i tuoi progetti MAUI!**

```
// Fine della presentazione – un semplice saluto dall'app
await Application.Current.MainPage.DisplayAlert("Fine", "Grazie per
l'attenzione e buon lavoro con MAUI!", "OK");
```

---

1 5 14 15 Creare app di Windows con .NET MAUI - Windows apps | Microsoft Learn
https://learn.microsoft.com/it-it/windows/apps/windows-dotnet-maui/

2 3 4 Corso MAUI: app cross-platform senza confini
https://sviluppatoremigliore.com/corsi/corso-maui

6 7 Build your first .NET MAUI app - .NET MAUI | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/maui/get-started/first-app?view=net-maui-10.0

8 9 10 11 12 13 Deployment & testing - .NET MAUI | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/maui/deployment/?view=net-maui-10.0