XAMARIN FORMS

Cross-Platform Mobile Applications Development

MAY 21, 2021

OVIDIU MOLDOVAN

IAISC 31212

CONȚINUT

XAMARIN FORMS

INTRODUCERE ÎN APLICAȚII MOBILE

Sistemele de operare cele mai folosite pe dispozitivele mobile sunt Android și iOS, cele două având o pondere pe piață, cumulate, de aproximativ 99.8%. Cele doua sunt foarte diferite ca structură, iar pentru dezvoltarea unei aplicații trebuie decis în primul rând limbajul de programare ce va fi folosit. În funcție de asta putem avea chiar și cicluri de viață diferite ale aplicației. [1] prezintă detalii despre procesul de dezvoltare specific fiecărei platforme.

Android este un sistemul de operare creat de Google, open-source, bazat pe Linux. Limbajul de programare folosit era în principal Java, dar un studiu [2] a concluzionat că inginerii software considerau procesul de dezvoltare pe Android cu 30% decât cel pe iOS. Google a ascultat cu atenție criticile comunității, iar in 2017 Kotlin a devenit unul dintre limbajele oficiale suportate de Android. Rulând pe aceeași mașină virtuală JVM și având interoperabilitate cu Java, Kotlin a fost rapid adoptat. Mai mult, acesta aduce și noi funcționalități. În prezent, Google susține ambele limbaje pentru a fi folosite în platforma lor prin mediul de dezvoltare Android Studio, bazat pe IntelliJ (JetBrains).

[1] prezintă în continuare și modalitatea iOS al celor de la Apple. Este menționată superioritatea sistemului de operare față de Android când vine vorba de securitate: magazinul de aplicații App Store este singura cale de a instala oficial o aplicație, iar aplicațiile prezente trebuie să treacă de un proces de aprobare mult mai riguros decât cel al Google Play Store. Limbajul de programare folosit obișnuia să fie Objective-C, dar în 2014 Apple a introdus Swift, un limbaj nou menit să scurteze timpul necesar dezvoltării unei aplicații. Acesta a introdus câteva similarități cu limbaje deja populare: JavaScript, Python, C# sau Java, iar vitezele pot fi comparate cu cele ale C++. De asemenea, Swift poate rula cod scris în Objective-C, C sau C++, motiv pentru care a avut o rată de adopție foarte bună. La fel ca în cazul Android, Apple oferă suport pentru ambele limbaje în mediul de dezvoltare Xcode. Din păcate, Xcode poate rula doar pe un dispozitiv cu sistemul de operare macOS, dar Swift este un limbaj open-source ce poate rula, cel puțin în teorie, și pe sisteme Windows sau Linux.

Totuși, în foarte multe cazuri este nevoie ca o aplicație mobilă să ruleze pe ambele platforme. Se propune astfel următoarea clasificare a aplicațiilor mobile:

- Aplicații **Native**: scrise folosind limbajele de programare oficiale ale fiecărui sistem de operare. În acest fel au acces direct la toate funcțiile sistemului de operare și la bibliotecile respective. Dezvoltarea nativă va rezulta în aplicații mai eficiente.
- Aplicații **Web**: scrise folosind tehnologiile web (HTML, CSS, JavaScript). Vor avea toate limitările unui browser deoarece folosesc browserul integrat în sistemul de operare.
- Aplicații **Hibrid**: scrise folosind limbaje cu o utilitate mai generală, cum ar fi **C# (Xamarin)**, Python (Flutter) sau JavaScript (React Native). Acestea se pot apropia ca eficiență de aplicațiile native, dar pot să nu includă toate funcțiile.

Asta poate fi foarte clar pe iOS, unde Steve Jobs a anunțat, odată cu primul iPhone în 2007, că acesta suportă Aplicații Web, iar suportul pentru Aplicațiile Native a venit un an mai târziu, odată cu lansarea App Store [3]. Totuși, pe Android situația poate sta puțin diferit deoarece Google oferă

suport oficial pentru C/C++ prin ceea ce numește Android NDK (Native Development Kit). Oricum, ambele sisteme rulează nativ C/C++, lucru folosit în special de platformele de dezvoltare de jocuri: Unity, Unreal Engine. Astfel, se propune încă o clasificare după cum urmează:

- Aplicații **Endemice**: aplicațiile create folosind uneltele oficiale (SDK – Source Development Kits).
- Aplicații **Pandemice**: aplicațiile create folosind limbajele ce rulează pe sistemul de operare fără unelte oficiale (C/C++, HTML, CSS, JavaScript).
- Aplicații **Ecdemice**: aplicațiile create folosind tool-uri multi-platformă (**Xamarin**, Flutter, React Native) dar translatate în codul mașină specific sistemului de operare pe care rulează (Low Level Virtual Machine – LLVM pe iOS și Android Runtime – ART pe Android).

Autorul articolului este de acord că denumirile pot fi îmbunătățite. Cu toate astea, inginerii software trebuie să decidă limbajul de programare și uneltele folosite în funcție de necesitățile proiectului și de experiența lor cu diferitele limbaje. De exemplu, un programator cu experiență în JavaScript ar putea alege foarte ușor să realizeze o aplicație în React Native, principiile fiind foarte similare cu biblioteca ReactJS.

În cele ce urmează voi prezenta Xamarin Forms, una dintre metodele populare de realizare a aplicațiilor multi-platformă.

CE ESTE XAMARIN?

Xamarin este o parte a .NET Framework ce acoperă aplicațiile mobile. Cum am spus și anterior, principalul motiv pentru a folosi Xamarin este capacitatea acestuia de a genera aplicații ce rulează pe mai multe platforme folosind același cod de bază. Desigur, are și alte avantaje. Unul dintre ele ar fi faptul că limbajul folosit este C#, un limbaj de uz general ce nu se folosește numai pentru aplicații mobile, față de Swift sau Kotlin. Un alt avantaj este reprezentat de bibliotecile numeroase deja disponibile pentru .NET.

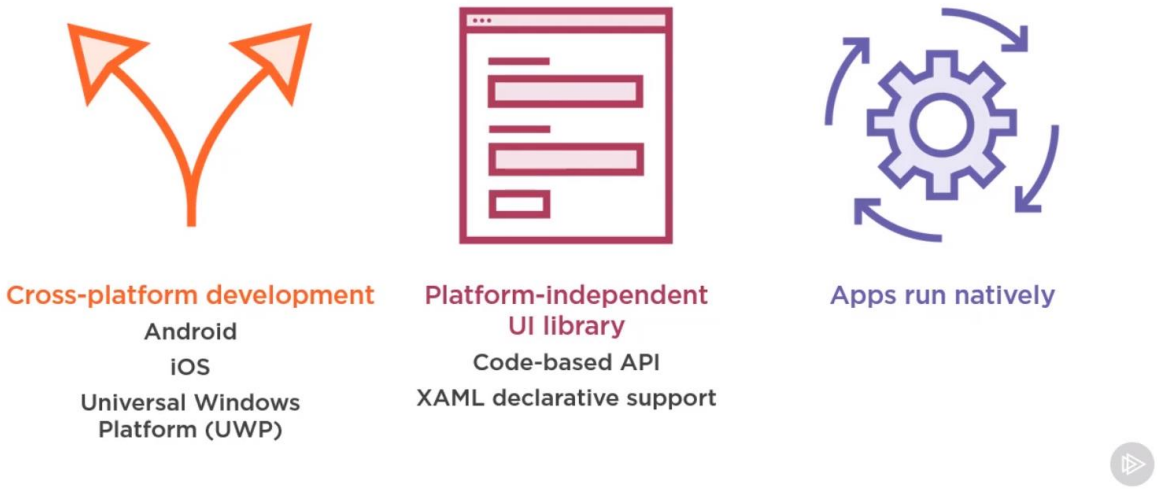

Figura 1 – Avantaje ale Xamarin [4]

Figura 1 mai prezintă faptul că se generează o interfață diferită în funcție de platformă și de standardele de design impuse la nivel de sistem de operare. O comparație între cele două poate fi vazută în Figura 2. De asemenea, este important de menționat că, deși Android-ul este mai permisiv, aprobarea în App Store de către Apple depinde și de design.

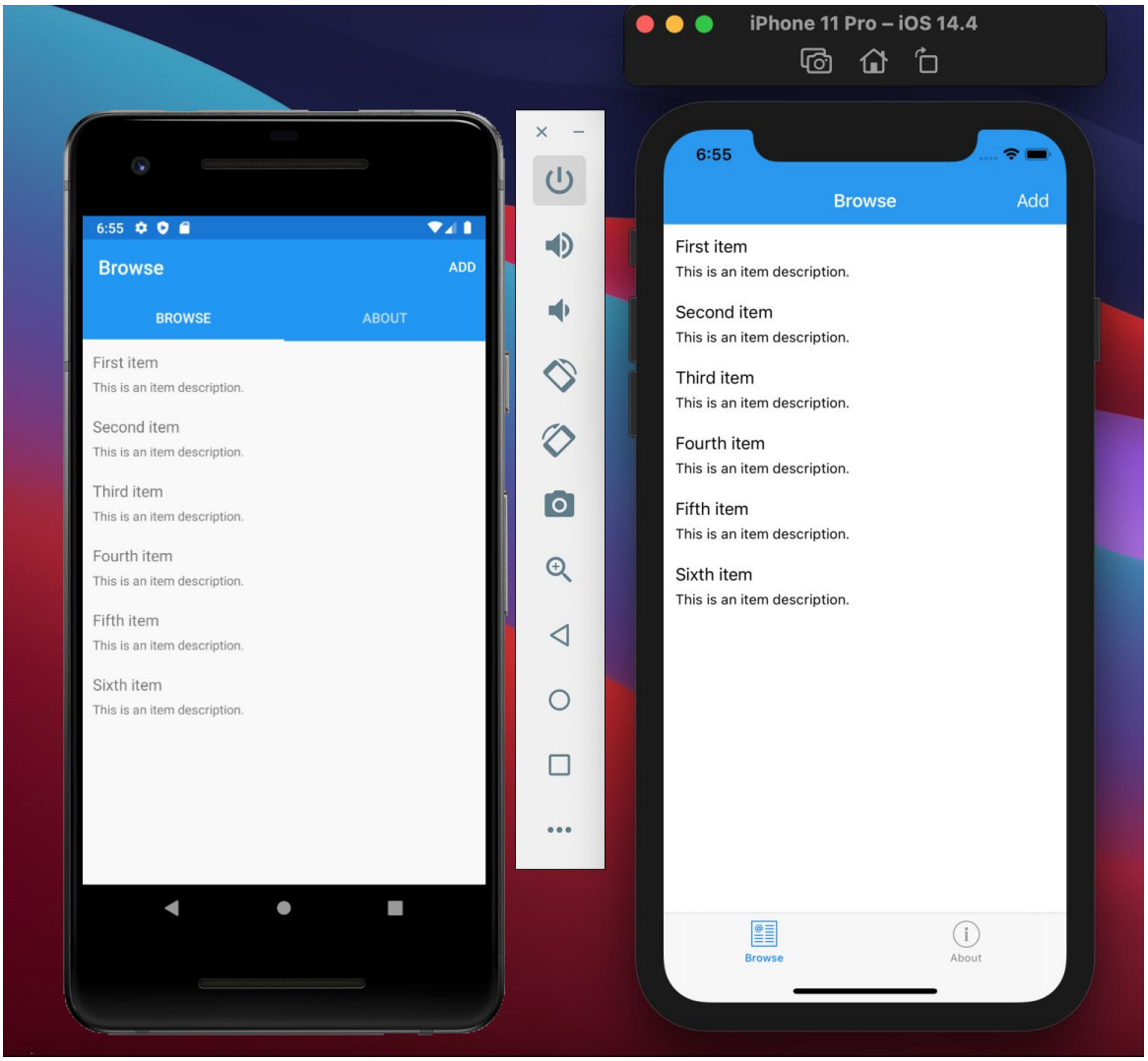

Figura 2 – Comparație design între aplicațiile iOS și Android.
Remarcăm poziția butoanelor de selectare a paginilor, poziția titlurilor sau chiar butonul *Add* care este ușor diferit.

Codul generat rulează, în final, nativ pe fiecare platformă suportată. Figura 3 prezintă structura unui proiect Xamarin Forms.

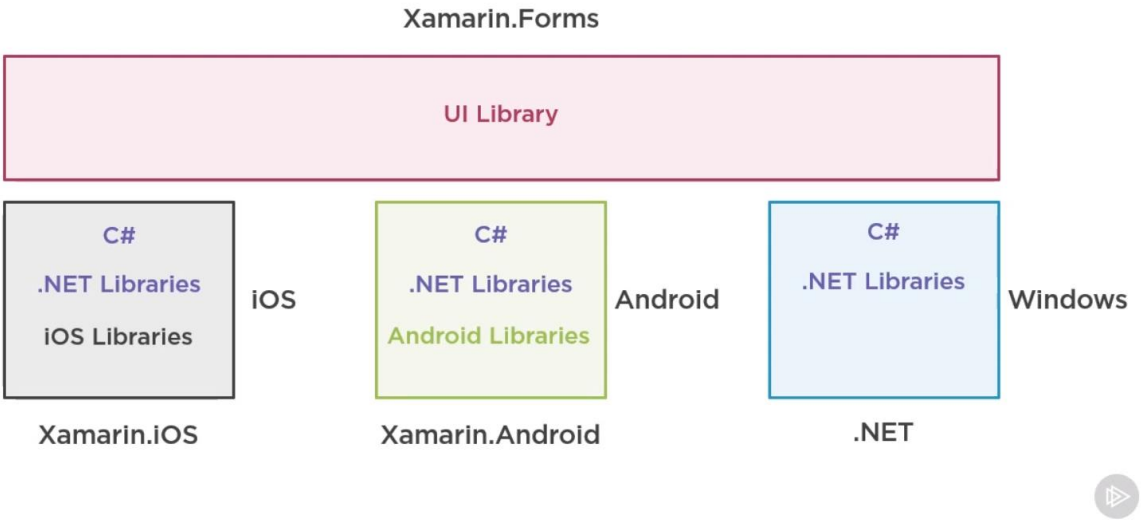

Figura 3 – Structura proiectului [4]

MEDIU DE DEZVOLTARE. COMPATIBILITATE

Fiind o parte a ecosistemului .NET, mediul de dezvoltare (IDE – Integrated Development Environment) cu suport oficial este Visual Studio. Alte soluții precum Rider (JetBrains) sunt de asemenea folosite, .NET fiind până la urmă open-source.

Pentru a dezvolta aplicații pentru Windows sau Android prin Xamarin Forms avem nevoie de sistemul de operare Windows, iar pentru a dezvolta aplicații pe iOS și Android de sistemul de operare macOS. Asta este din cauză că Xamarin folosește funcții ale sistemelor de operare, iar unele dintre acestea au suport oficial doar pe platforma companiei deținătoare. Visual Studio pentru Windows se poate folosi de un simulator iOS atâta timp cât putem oferi acces la distanță unui dispozitiv Mac din rețea. Altfel, aplicația nu poate fi testată pe iOS, cel puțin folosind resursele recomandate.

Pentru a acoperi ambele platforme este de preferat un sistem Mac. Exemplele prezentate în continuare au fost rulate pe varianta de macOS a Visual Studio 2019.

INTERFAȚA DE UTILIZATOR. XAML

STRUCTURA INTERFEȚEI DE UTILIZATOR

Figura 4 prezintă sumar structura interfeței de utilizator (User Interface – UI) într-o aplicație Xamarin.

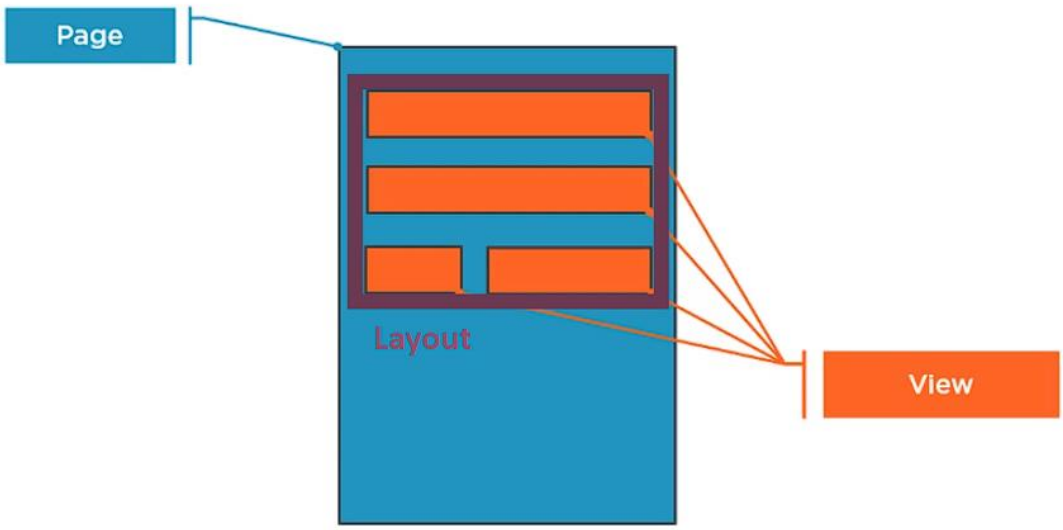

Figura 4 – Structura interfeței de utilizator [4]

Ecranele aplicației sunt denumite Page (pagini). Acestea pot fi, de exemplu, de tipul ContentPage (o pagină normală), MasterDetailPage (o pagină de meniu), NavigationPage (o pagină la care s-a ajuns dintr-o altă pagină, cu gesturi și butoane implicite pentru a merge înapoi) sau TabbedPage (împărțirea conținutului pe taburi, cum s-a văzut anterior în Figura 2).

Elementele de pe o pagină se numesc View. Acestea pot fi, de exemplu, de tipul Label (etichetă text), Input (câmp de intrare), Button, Picker (selectoare de liste, date).

Pentru organizarea elementelor în pagini se folosesc Layout-uri. Un Layout este un grup care conține mai multe View-uri. Avantajul folosirii este stilizarea comună (de exemplu, așezarea în pagină a câmpurilor dintr-un formular). O pagină poate conține mai multe Layout-uri.

XAML

Interfața aplicațiilor Xamarin se scrie cu ajutorul XAML (Extensible Application Markup Language). XAML este bazat pe limbajul XML și este asemănător cu acesta din urmă sau cu alte limbaje de marcare (de exemplu HTML). Este susținut de Microsoft și integrat în multe dintre funcționalitățile .NET cum ar fi și Windows Presentation Forms (WPF).

În Figura 5 este prezentat un exemplu cu câteva câmpuri și modul în care arată acestea la compilarea aplicației pe telefon.

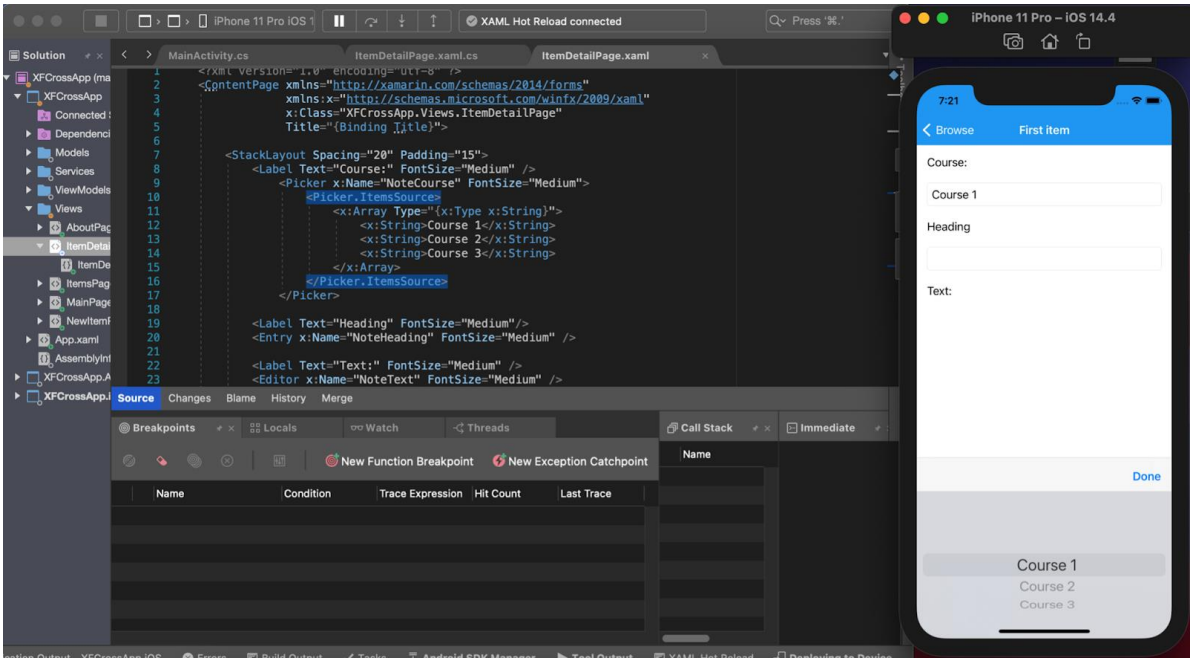

Figura 5 – Exemplu XAML

Versiunile mai vechi de Xamarin ofereau un previzualizator pentru codul scris în XAML pentru a oferi programatorului o idee despre cum va arăta design-ul în final. Versiunile noi folosesc funcția XAML Hot Reload. Cât timp compilatorul este conectat la aplicația din simulator sau emulator, interfața se va schimba, în timp real, la fiecare salvare a fișierelor XAML, iar modificările se vor vedea imediat.

Fiecare fișier XAML are asociat o clasă în C# ce se poate ocupa cu logica paginii respective. În Figura 5 se poate observa și atributul XAML *x:Name* al câmpurilor de intrare. Acesta ne dă acces la acel câmp în fișierul C# asociat (Figura 6).

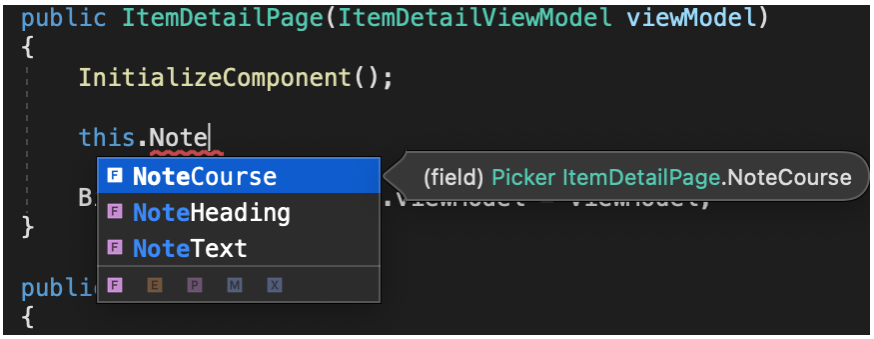

Figura 6 – Accesul în C# la câmpurile denumite în XAML

Similar, pentru a apela o metodă o vom declara prin atributul *Clicked* (Figura 7).

```
<ContentPage.ToolbarItems>
    <ToolbarItem Text="Cancel" Clicked="Cancel_Clicked"/>
    <ToolbarItem Text="Save" Clicked="Save_Clicked"/>
</ContentPage.ToolbarItems>
```

Figura 7 – Atributul *Clicked* in XAML

Cu același nume vom declara o metodă în fișierul C# (Figura 8) ce se va ocupa de evenimentul respectiv (de exemplu, Click pe buton).

```
void Save_Clicked(object sender, EventArgs e)
{
    DisplayAlert("Save option", "Save was selected", "Button 1", "Button 2");
}

void Cancel_Clicked(object sender, EventArgs e)
{
    DisplayAlert("Cancel option", "Cancel was selected", "Button 1", "Button 2");
}
```

Figura 8 – Metodele asociate evenimentelor de Click

Astfel, la click pe butoanele Cancel sau Save se vor afișa alertele definite în cod. Codul rulat este prezentat în Figura 9. Remarcăm și aici modul diferit al afișării unei alerte, specific sistemului de operare.

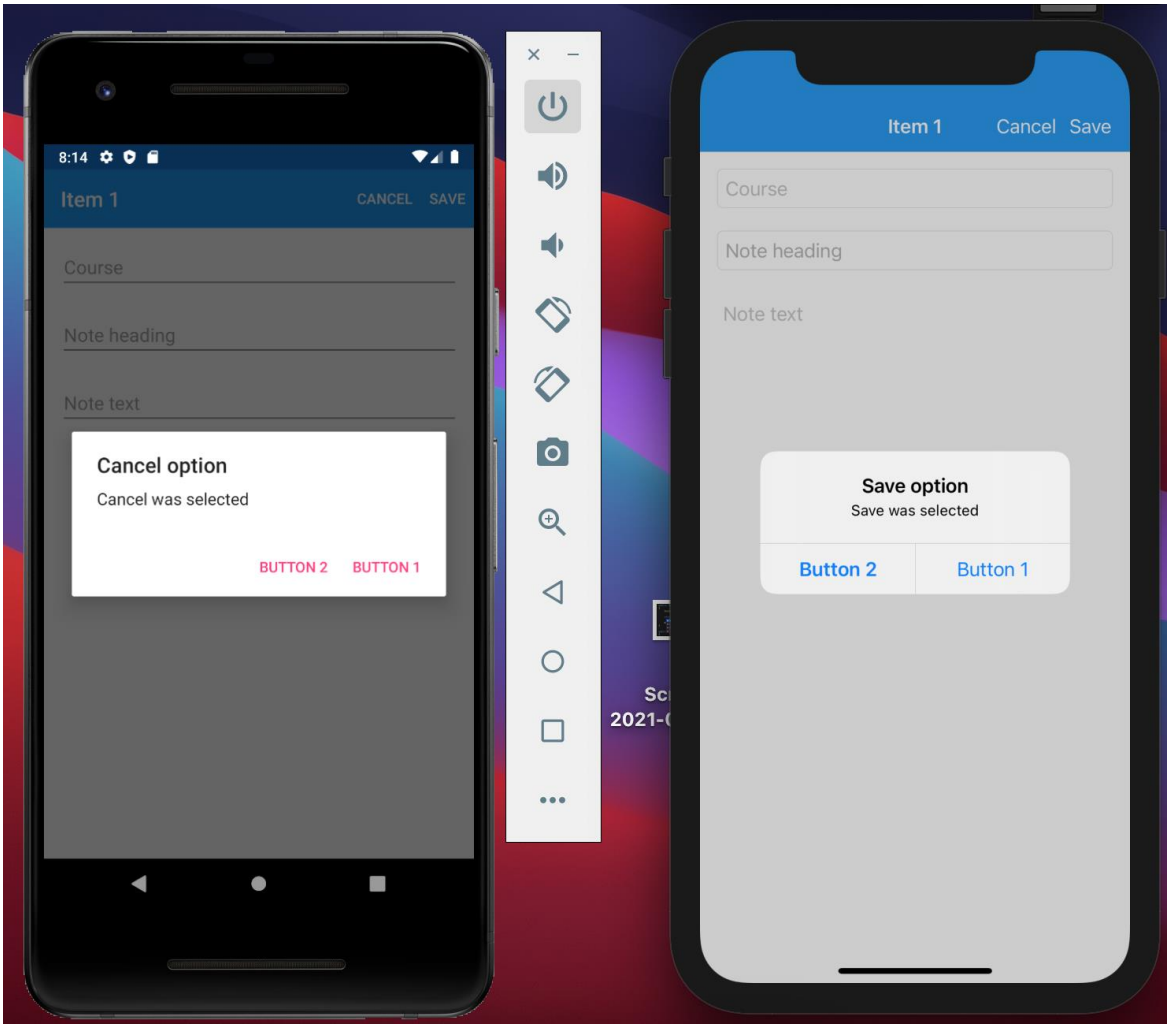

Figura 9 – Mesajele de alertă

DATA BINDING

Data Binding este un șablon de proiectare (design pattern) ce are ca scop eficientizarea conexiunii dintre interfața cu utilizatorul și modelul de date din cod. În XAML specificăm pentru câmpurile dorite atributul *Binding* (Figura 10).

```
<Entry Text="{Binding Heading}"
       x:Name="NoteHeading"
       FontSize="Medium"
       Placeholder="Note heading"/>

<Editor Text="{Binding Text}"
        x:Name="NoteText"
        FontSize="Medium"
        Placeholder="Note text"
        AutoSize="TextChanges"/>
```

Figura 10 – Data Binding în XAML

În codul asociat paginii trebuie să setăm, în constructor, atributul *BindingContext* ca fiind obiectul cu care dorim să lucrăm (Figura 11).

```
public ItemDetailPage()
{
    InitializeComponent();

    InitializeData();

    BindingContext = Note;
}

void InitializeData()
{
    Note = new Note
    {
        Heading = "Test Note",
        Text = "Text test",
    };
}
```

Figura 11 – Data Binding în C#

Astfel avem acces la valorile *Heading* și *Text* ale obiectului *Note*. Mai mult, procedeul Data Binding creeaza și o conexiune de intrare și una de ieșire (2-way data binding), astfel că valorile din cod sunt transmise interfeței, iar valorile din interfață sunt transmise în cod.

MODEL-VIEW-VIEWMODEL (MVVM)

MVVM nu este neapărat specific Xamarin sau .NET, ci este o arhitectură întâlnită în multe alte aplicații sau limbaje. Principiul acestui șablon arhitectural (architectural pattern) este că stratul de prezentare nu ar trebui să aiba informații despre stratul de date și vice-versa. Astfel introducem un strat intermediar denumit ViewModel. Acesta ia datele din stratul Model și le prelucrează pentru stratul View.

Xamarin oferă funcționalitatea de bază BaseViewModel (Figura 12), celelalte clase tip ViewModel extinzând-o pe aceasta.

```
public class ItemDetailViewModel : BaseViewModel
{
    public Note Note { get; set; }
    public ItemDetailViewModel(Item item = null)
    {
        Title = item?.Text;
        Note = new Note
        {
            Heading = "test Note",
            Text = "test mvvm"
        };
    }
}
```

Figura 12 – Declararea unui ViewModel

La nivelul paginii vom face Binding pe ViewModel-ul creat (Figura 13).

```
public ItemDetailPage()
{
    InitializeComponent();

    InitializeData();

    viewModel = new ItemDetailViewModel();
    BindingContext = viewModel;
    NoteCourse.BindingContext = this;
}
```

Figura 13 – Binding pe ViewModel

Putem crea apoi o nouă valoare în ViewModel care să returneze doar ceea ce ne dorim (Figura 14), iar in XAML vom face Binding pe acea valoare nou definită (Figura 15).

```
public String NoteHeading
{
    get { return Note.Heading; }
    set
    {
        Note.Heading = value;
        OnPropertyChanged();
    }
}
```

Figura 14 – Definirea unei noi valori în ViewModel cu accesori *get* și *set* specifici

```
<Entry Text="{Binding NoteHeading}"
```

Figura 15 – Binding în XAML pe noua valoare

Observăm în Figura 14 apelarea metodei *OnPropertyChanged*. Acest lucru este posibil deoarece clasa de bază BaseViewModel implementează interfața *INotifyPropertyChanged*. Apelarea metodei face ca interfața să se reîmprospăteze și să afișeze noua valoare setată. Exemplul rulat se poate vedea în Figura 16.

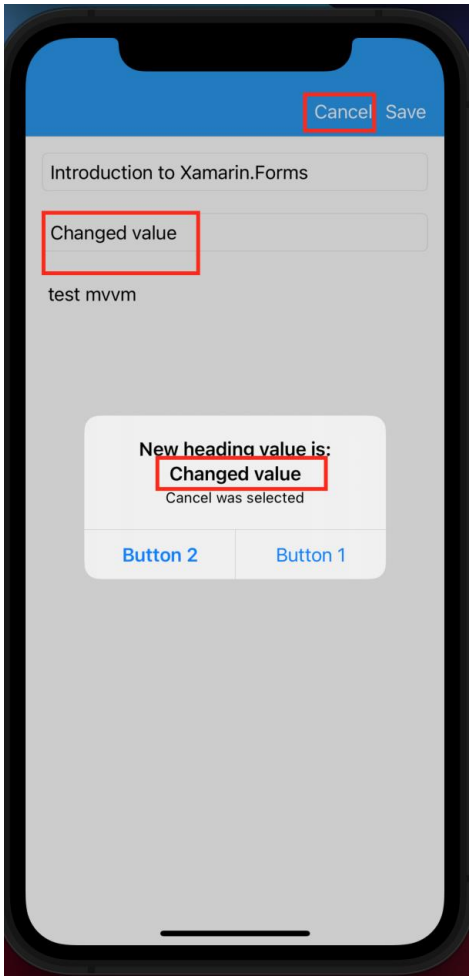

Figura 16 – Exemplul MVVM rulat

ALTE PROPRIETATI UTILE ALE XAML

Observăm în Figura 17 folosirea unui RefreshView cu o comandă de Load. Această acțiune se declanșează, de regulă, la executarea de către utilizator a unui gest de refresh (o glisare în sus – swipe up – pe pagina respectivă).

De asemenea putem observa, în aceeași figură, eticheta *TapGestureRecogniser* cu atributul *Tapped* ce duce către o metodă.

```
<RefreshView IsRefreshing="{Binding IsBusy, Mode=TwoWay}" Command="{Binding LoadItemsCommand}">
    <CollectionView x:Name="ItemsCollectionView"
            ItemsSource="{Binding Notes}">
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <StackLayout Padding="10">
                    <Label Text="{Binding Course}"
                           LineBreakMode="NoWrap"
                           Style="{DynamicResource ListItemTextStyle}"
                           FontSize="16" />
                    <Label Text="{Binding Heading}"
                           LineBreakMode="NoWrap"
                           Style="{DynamicResource ListItemDetailTextStyle}"
                           FontSize="13" />
                    <StackLayout.GestureRecognizers>
                        <TapGestureRecognizer NumberOfTapsRequired="1"
                                              Tapped="OnItemSelected">
                        </TapGestureRecognizer>
                    </StackLayout.GestureRecognizers>
                </StackLayout>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</RefreshView>
```

Figura 17 – Exemplu XAML

Metoda OnItemSelected (Figura 18) primește pe obiectul *sender* detaliile despre elementul care a accesat metoda respectivă.

```
async void OnItemSelected(object sender, EventArgs args)
{
    var layout = (BindableObject)sender;
    var note = (Note)layout.BindingContext;
    if (note == null) return;
    await Navigation.PushAsync(new ItemDetailPage(new ItemDetailViewModel(note)));
}

async void AddItem_Clicked(object sender, EventArgs e)
{
    await Navigation.PushModalAsync(new NavigationPage(new ItemDetailPage()));
}
```

Figura 18 – Exemplu C#

De asemenea, observăm în Figura 18 două modalități diferite de navigare: Push (Async) și PushModal (Async). Diferența dintre cele două este, în principal, faptul că PushAsync va crea o pagină în dreapta, iar o glisare la stânga va fi echivalentă cu apăsarea butonului Back (Înapoi). PopModalAsync va deschide o nouă pagină de jos în sus, iar pentru a o închide trebuie să apelăm metoda PopModalAsync (Figura 19).

```
void Cancel_Clicked(object sender, EventArgs e)
{
    Navigation.PopModalAsync();
}
```

Figura 19 – Metoda PopModalAsync

PUBLISH ȘI SUBSCRIBE

Xamarin oferă o implementare a mecanismului de tip Publish / Subscribe prin clasa MessagingCenter.

Prin metoda Send (Figura 20) putem trimite un mesaj, de exemplu de la View, către ViewModel pentru a separa funcționalitatea de vizualizare și cea de prelucrare a datelor. Pentru a specifica tipul mesajului este necesar să definim un string unic pentru acea funcționalitate.

```
void Save_Clicked(object sender, EventArgs e)
{
    MessagingCenter.Send(this, "SaveNote", viewModel.Note);

    Navigation.PopModalAsync();
}
```

Figura 20 – Funcționalitatea de tip Publish

Pentru a executa o acțiune când un astfel de eveniment este trimis folosim metoda Subscribe (Figura 21) cu același string definit în Send.

```
Title = "Browse";
Notes = new ObservableCollection<Note>();
LoadItemsCommand = new Command(async () => await ExecuteLoadItemsCommand());

MessagingCenter.Subscribe<ItemDetailPage, Note>(this, "SaveNote",
    async (sender, note) => {
        Notes.Add(note);
        await PluralsightDataStore.AddNoteAsync(note);
});
```

Figura 21 – Funcționalitatea de tip Subscribe

LUCRATUL CU HTTP

HTTP SAU HTTPS

Lucrând cu un API local creat de mine, nu dispun de certificat HTTPS cât timp nu public aplicația. Deși *localhost* funcționează cu HTTPS, trecerea de la *localhost* la accesarea API-ului prin IP forțează obținerea unui certificat sau folosirea HTTP. Folosirea HTTP este necesară pentru a testa aplicațiile mobile.

Android și iOS forțează în mod implicit folosirea URL-urilor securizate. Pentru a dezactiva asta trebuie să schimbăm setările proiectelor, după cum urmează:

a) Pe Android trebuie să adăugăm ultima linie din Figura 22 în fișierul AssemblyInfo.cs specific proiectului

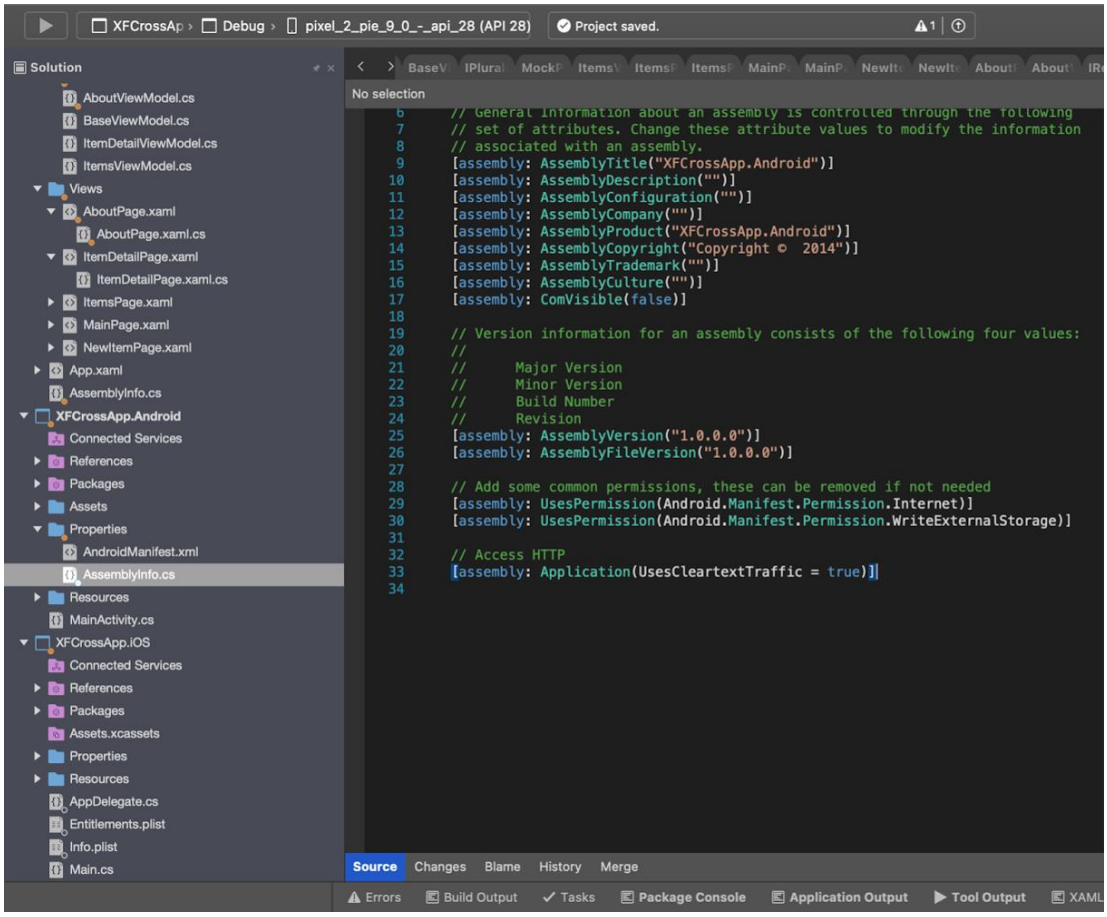

Figura 22 – Permiterea conexiunii HTTP pe Android

b) Pe iOS – Figura 23 – se setează din proprietățile proiectului opțiunea HttpClient implementation pe Managed.

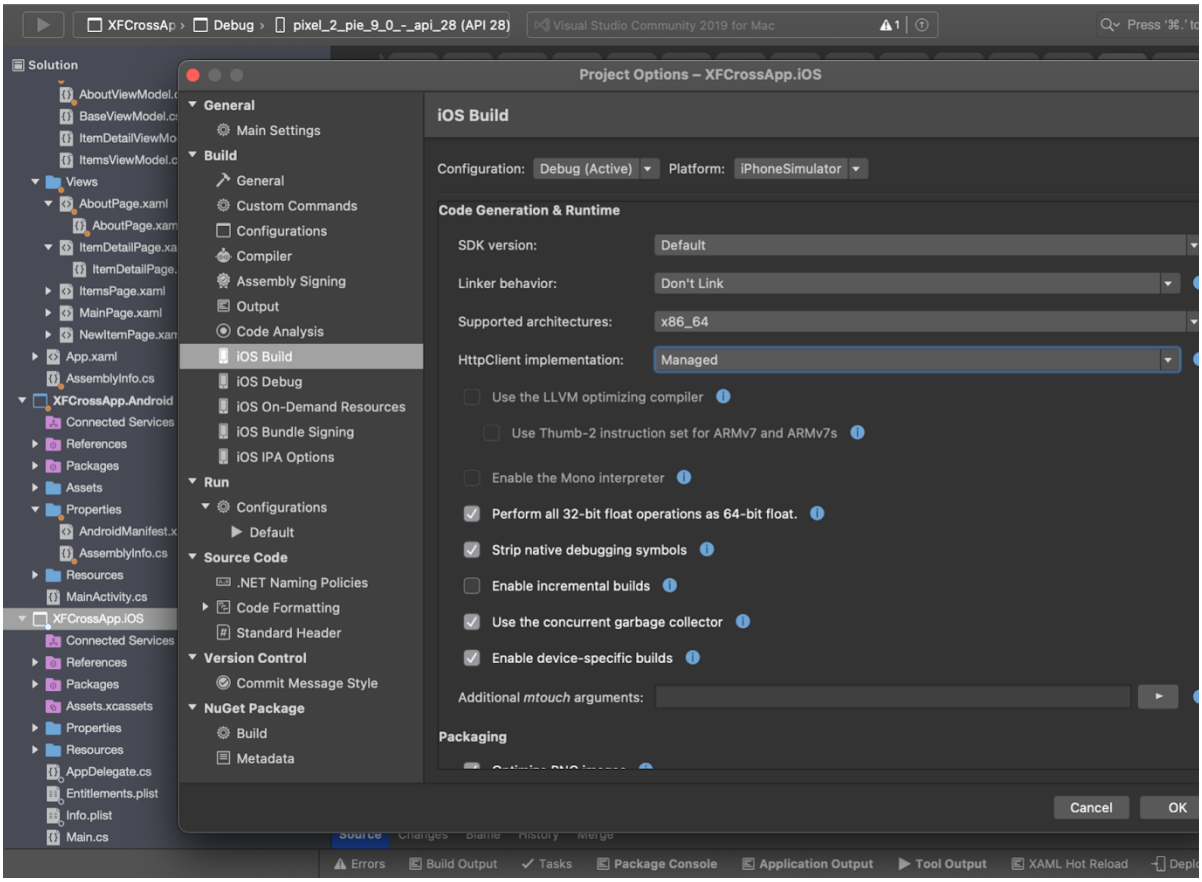

Figura 22 – Permiterea conexiunii HTTP pe iOS

RESTFUL SERVICES

GET

Pentru un exemplu de GET request s-a folosit funcționalitatea API-ului de a returna data și ora server-ului. Serviciul este prezentat în Figura 23.

```
public async Task<string> GetServerTimeAsync()
{
    Uri uri = new Uri(string.Format(Constants.ApiURL + "/Api/General/hour",
        string.Empty));

    string result = "No response";
    HttpResponseMessage response;

    try
    {
        response = await httpClient.GetAsync(uri);
    }
    catch(Exception ex)
    {
        Debug.WriteLine(ex);
        response = new HttpResponseMessage {
            StatusCode = System.Net.HttpStatusCode.InternalServerError
        };
    }
    if (response.IsSuccessStatusCode)
    {
        string content = await response.Content.ReadAsStringAsync();
        result = JsonSerializer.Deserialize<string>(content);
    }
    return result;
}
```

Figura 23 – Serviciul de GetServerTime

Rezultatul este afișat într-un Label, similar metodelor prezentate anterior.

POST

Pentru demonstrarea funcționalității POST s-a realizat funcția de Login a API-ului. Serviciul este prezentat în Figura 24.

```
public async Task<string> LoginAsync(Login login)
{
    Uri uri = new Uri(string.Format(Constants.ApiURL + "/Api/User/login",
        string.Empty));

    HttpContent result = null;
    HttpResponseMessage response;

    string json = JsonSerializer.Serialize<Login>(login);
    StringContent content = new StringContent(json, Encoding.UTF8, "application/json");

    try
    {
        response = await httpClient.PostAsync(uri, content);
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex);
        response = new HttpResponseMessage
        {
            StatusCode = System.Net.HttpStatusCode.InternalServerError
        };
    }
    if (response.IsSuccessStatusCode)
    {
        result = response.Content;
    }
    return await result?.ReadAsStringAsync();
}
```

Figura 24 – Serviciu de Login

Rezultatul este apoi deserializat (Figura 25), iar Modelul Login primește un atribut DisplayName – numele asociat unui utilizator și un Token folosit pentru autorizare.

```
async Task ExecuteLogin()
{
    IsBusy = true;
    string result = "";

    try
    {
        result = await restService.LoginAsync(new Login
        {
            Username = Username,
            Password = Password,
        });
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex);
    }
    finally
    {
        Login = JsonConvert.DeserializeObject<Login>(result) ?? new Login();
        if (Login != null)
        {
            DisplayName = Login?.Name ?? "something failed";
        }
        IsBusy = false;
    }
}
```

Figura 25 – Deserializarea răspunsului de Login

Pentru deserializare s-a folosit un pachet NuGet disponibil în platforma .NET.

APLICAȚIA DEMONSTRATIVĂ

Aplicația demonstrativă (Figura 26) are, în final, două tab-uri. Primul tab reprezintă lucrul cu o listă de obiecte din interiorul aplicației. Lista poate fi vizualizată, iar elementele ei pot fi modificate sau se pot adăuga altele noi.

Al doilea tab reprezintă conexiunea cu API-ul local: Login-ul care afișează, în cazul în care s-a efectuat corect, DisplayName-ul asociat utilizatorului. De asemenea se poate apăsa butonul *Get Date* pentru a afișa ora server-ului.

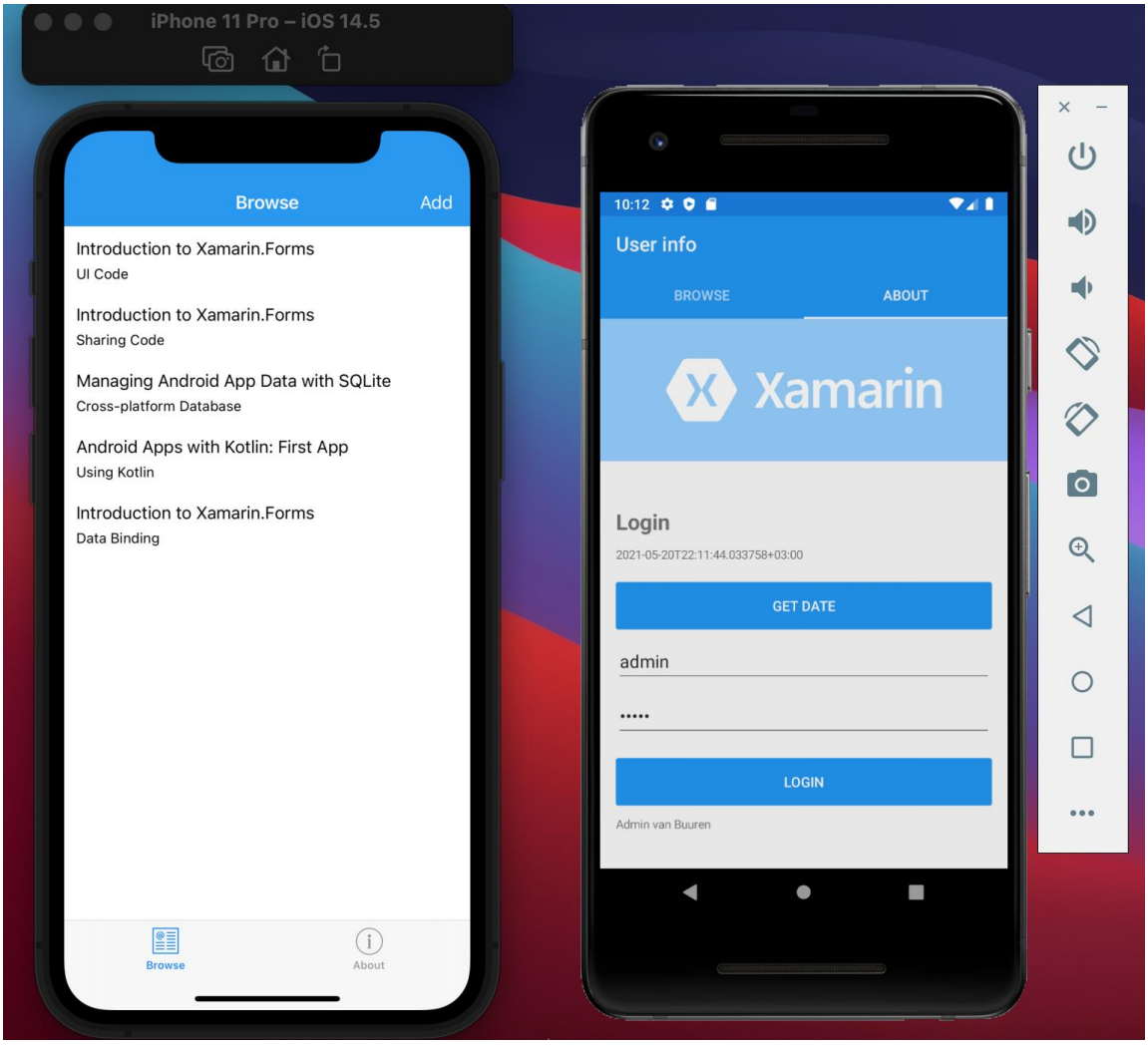

Figura 26 – Aplicația demonstrativă funcționând pe un dispozitiv iOS (stânga) și unul Android (dreapta)

REFERINȚE

[1] H. Brito, A. Gomes, Á. Santos și J. Bernardino, „JavaScript in mobile applications: React native vs ionic vs NativeScript vs native development,” *2018 13th Iberian Conference on Information Systems and Technologies (CISTI),* pp. 1-6, 2018.

[2] V. Oliveira, L. Teixeira și F. Ebert, „On the Adoption of Kotlin on Android Development: A Triangulation Study,” *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER),* pp. 206-2016, 2020.

[3] R. Nunkesser, „Beyond Web/Native/Hybrid: A New Taxonomy for Mobile App,” *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft),* pp. 214-218, 2018.

[4] J. Wilson, „Introduction to Xamarin.Forms,” PluralSight, 13 May 2019. [Interactiv]. Available: https://app.pluralsight.com/library/courses/introduction-xamarin-forms/table-of-contents. [Accesat 18 05 2021].