EBook Gratis

APRENDIZAJE
xamarin

Free unaffiliated eBook created from
Stack Overflow contributors.

#xamarin

Tabla de contenido

Acerca de 1

Capítulo 1: Empezando con Xamarin 2

Observaciones 2

Examples 2

Instalando Xamarin Studio en OS X 2

Proceso de instalación 4

Próximos pasos 5

Hola mundo usando Xamarin Studio: Xamarin.Forms 5

Capítulo 2: Código compartido entre proyectos 7

Examples 7

El patrón de puente 7

El patrón de localización de servicios 8

Capítulo 3: Validación de objetos por anotaciones 11

Introducción 11

Examples 11

Ejemplo simple 11

Creditos 13

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: xamarin

It is an unofficial and free xamarin ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official xamarin.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Xamarin

Observaciones

Esta sección proporciona una descripción general de qué es xamarin y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de xamarin, y vincular a los temas relacionados. Dado que la Documentación para xamarin es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Examples

Instalando Xamarin Studio en OS X

El primer paso para iniciar el desarrollo de Xamarin en una máquina OS X, es descargar e instalar la versión Xamarin Studio Community desde el sitio web oficial . Es necesario rellenar algunos campos para descargar el instalador como se muestra en la siguiente imagen.

Xamarin Products Custom

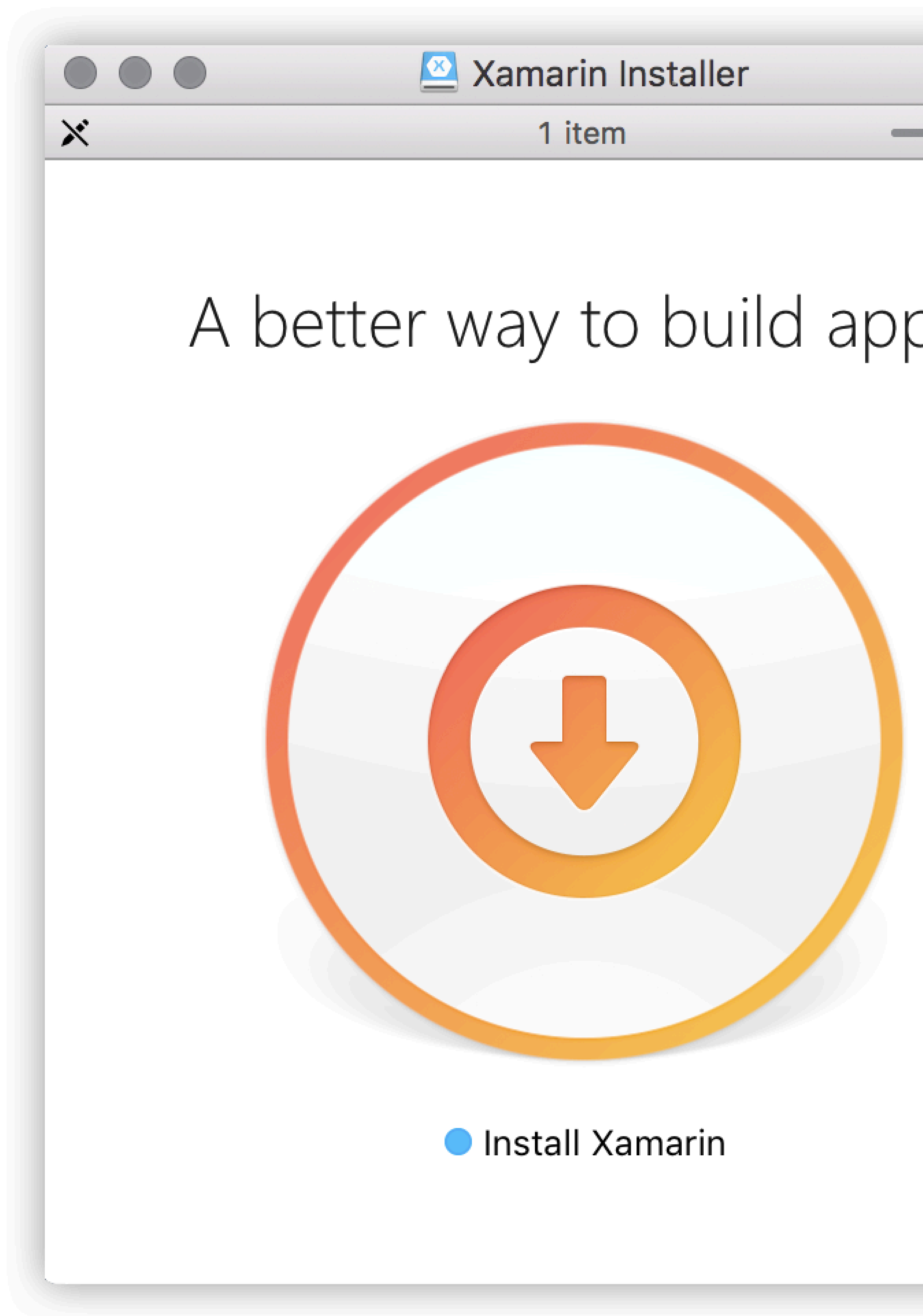
Dow

Nice! You are about to d
C# and sha

- La última versión de Xcode de la Mac App Store o el sitio web de desarrolladores de Apple .
  sitio web de desarrolladores de Apple .
- Mac OS X Yosemite (10.10) y superior

Proceso de instalación

Una vez que se cumplen los requisitos previos, ejecute el instalador de Xamarin haciendo doble clic en el logotipo de Xamarin.



OS X Gatekeeper puede mostrar un cuadro de diálogo que le solicita una confirmación para abrir

https://riptutorial.com/es/xamarin/topic/899/empezando-con-xamarin

Capítulo 2: Código compartido entre proyectos

Examples

El patrón de puente

El patrón de puente es uno de los patrones de diseño de inversión de control más básicos. Para Xamarin, este patrón se usa para hacer referencia al código dependiente de la plataforma desde un contexto independiente de la plataforma. Por ejemplo: usar AlertDialog de Android desde una biblioteca de clases portátil o formularios Xamarin. Ninguno de esos contextos sabe qué es un objeto AlertDialog, por lo que debe envolverlo en un cuadro para que lo utilicen.

```
// Define a common interface for the behavior you want in your common project (Forms/Other
PCL)
public interface IPlatformReporter
{
    string GetPlatform();
}


// In Android/iOS/Win implement the interface on a class
public class DroidReporter : IPlatformReporter
{
    public string GetPlatform()
    {
        return "Android";
    }
}


public class IosReporter : IPlatformReporter
{
    public string GetPlatform()
    {
        return "iOS";
    }
}


// In your common project (Forms/Other PCL), create a common class to wrap the native
implementations
public class PlatformReporter : IPlatformReporter
{
    // A function to get your native implemenation
    public static func<IPlatformReporter> GetReporter;

    // Your native implementation
    private IPlatformReporter _reporter;

    // Constructor accepts native class and stores it
    public PlatformReporter(IPlatformReporter reporter)
    {
```

```
        _reporter = GetReporter();
    }

    // Implement interface behavior by deferring to native class
    public string GetPlatform()
    {
        return _reporter.GetPlatform();
    }
}


// In your native code (probably MainActivity/AppDelegate), you just supply a function that
returns your native implementation
public class MainActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.activity_main);

        PlatformReporter.GetReporter = () => { return new DroidReporter(); };
    }
}


public partial class AppDelegate : UIApplicationDelegate
{
    UIWindow window;

    public override bool FinishedLaunching(UIApplication app, NSDictionary options)
    {
        window = new UIWindow(UIScreen.MainScreen.Bounds);
        window.RootViewController = new UIViewController();
        window.MakeKeyAndVisible();

        PlatformReporter.GetReporter = () => { return new IosReporter(); };

        return true;
    }
}


// When you want to use your native implementation in your common code, just do as follows:
public void SomeFuncWhoCares()
{
    // Some code here...

    var reporter = new PlatformReporter();
    string platform = reporter.GetPlatform();

    // Some more code here...
}
```

El patrón de localización de servicios

El patrón de diseño del Localizador de Servicios es casi una inyección de dependencia. Al igual que el patrón de puente, este patrón se puede usar para hacer referencia al código dependiente de la plataforma desde un contexto independiente de la plataforma. Lo más interesante es que este patrón se basa en el patrón de singleton: todo lo que coloque en el localizador de servicios

será un singleton de facto.

```
// Define a service locator class in your common project
public class ServiceLocator {
    // A dictionary to map common interfaces to native implementations
    private Dictionary<object, object> _services;

    // A static instance of our locator (this guy is a singleton)
    private static ServiceLocator _instance;

    // A private constructor to enforce the singleton
    private ServiceLocator() {
        _services = new Dictionary<object, object>();
    }

    // A Singleton access method
    public static ServiceLocator GetInstance() {
        if(_instance == null) {
            _instance = new ServiceLocator();
        }

        return _instance;
    }

    // A method for native projects to register their native implementations against the
common interfaces
    public static void Register(object type, object implementation) {
        _services?.Add(type, implementation);
    }

    // A method to get the implementation for a given interface
    public static T Resolve<T>() {
        try {
            return (T) _services[typeof(T)];
        } catch {
            throw new ApplicationException($"Failed to resolve type: {typeof(T).FullName}");
        }
    }


//For each native implementation, you must create an interface, and the native classes
implementing that interface
public interface IA {
    int DoAThing();
}


public interface IB {
    bool IsMagnificent();
}


public class IosA : IA {
    public int DoAThing() {
        return 5;
    }
}


public class DroidA : IA {
    public int DoAThing() {
```

```
        return 42;
    }
}


// You get the idea...


// Then in your native initialization, you have to register your classes to their interfaces
like so:
public class MainActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.activity_main);

        var locator = ServiceLocator.GetInstance();
        locator.Register(typeof(IA), new DroidA());
        locator.Register(typeof(IB), new DroidB());
    }
}


public partial class AppDelegate : UIApplicationDelegate
{
    UIWindow window;

    public override bool FinishedLaunching(UIApplication app, NSDictionary options)
    {
        window = new UIWindow(UIScreen.MainScreen.Bounds);
        window.RootViewController = new UIViewController();
        window.MakeKeyAndVisible();

        var locator = ServiceLocator.GetInstance();
        locator.Register(typeof(IA), new IosA());
        locator.Register(typeof(IB), new IosB());

        return true;
    }
}


// Finally, to use your native implementations from non-native code, do as follows:
public void SomeMethodUsingNativeCodeFromNonNativeContext() {
     // Some boring code here


     // Grabbing our native implementations for the current platform
     var locator = ServiceLocator.GetInstance();
     IA myIA = locator.Resolve<IA>();
     IB myIB = locator.Resolve<IB>();


    // Method goes on to use our fancy native classes
}
```

Lea Código compartido entre proyectos en línea:
https://riptutorial.com/es/xamarin/topic/6183/codigo-compartido-entre-proyectos

Capítulo 3: Validación de objetos por anotaciones

Introducción

mvc.net introduce anotaciones de datos para la validación del modelo. Esto también se puede hacer en Xamarin.

Examples

Ejemplo simple

Agregar el paquete nuget `System.ComponentModel.Annotations`

Definir una clase:

```
public class BankAccount
{

   public enum AccountType
   {
       Saving,
       Current
   }

   [Required(ErrorMessage="First Name Required")]
   [MaxLength(15,ErrorMessage="First Name should not more than 1`5 character")]
   [MinLength(3,ErrorMessage="First Name should be more than 3 character")]
   public string AccountHolderFirstName { get; set; }

   [Required(ErrorMessage="Last Name Required")]
   [MaxLength(15,ErrorMessage="Last Name should not more than 1`5 character")]
   [MinLength(3,ErrorMessage="Last Name should be more than 3 character")]
   public string AccountHolderLastName { get; set; }

   [Required]
   [RegularExpression("^[0-9]+$", ErrorMessage = "Only Number allowed in AccountNumber")]
   public string AccountNumber { get; set; }

   public AccountType AcType { get; set; }
}
```

Definir un validador:

```
public class GenericValidator
{
    public static bool TryValidate(object obj, out ICollection<ValidationResult> results)
    {
        var context = new ValidationContext(obj, serviceProvider: null, items: null);
        results = new List<ValidationResult>();
        return Validator.TryValidateObject(
```

```
            obj, context, results,
            validateAllProperties: true
        );
    }
}
```

utilizar el validador:

```
var bankAccount = new BankAccount();
ICollection<ValidationResult> lstvalidationResult;

bool valid = GenericValidator.TryValidate(bankAccount, out lstvalidationResult);
if (!valid)
{
    foreach (ValidationResult res in lstvalidationResult)
    {
        Console.WriteLine(res.MemberNames +":"+ res.ErrorMessage);
    }

}
Console.ReadLine();
```

Salida generada:

```
First Name Required
Last Name Required
The AccountNumber field is required.
```

Lea Validación de objetos por anotaciones en línea:
https://riptutorial.com/es/xamarin/topic/9720/validacion-de-objetos-por-anotaciones

Creditos

S. No	Capítulos	Contributors
1	Empezando con Xamarin	Akshay Kulkarni, Community, Gil Sand, hankide, Joel Martinez, Marius Ungureanu, Sven-Michael Stübe, thomasvdb
2	Código compartido entre proyectos	kellen lask, valdetero
3	Validación de objetos por anotaciones	Niek de Gooijer