Bài: Multi Task với WPF trong C#

Xem bài học trên website để ủng hộ Kteam: Multi Task với WPF trong C#

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage How Kteam nhé!

Trong bài học này, Kteam sẽ Giới thiệu đến bạn **Khung WPF Multi Task và cách sử dụng** khung này trong lập trình Selenium, giả lập, liên quan đa luồng và cần giao diện.

Nội dung:

Để đọc hiểu bài này tốt nhất các bạn nên có kiến thức cơ bản về phần:

- C# CƠ BẢN
- C# NÂNG CAO

Trong bài này, ta sẽ cùng tìm hiểu các vấn đề:

- Giới thiệu phần mềm WPF Multi Task
- Cách sử dụng và tùy chỉnh khung

Source code tham khảo

MainWindow.xaml.cs

:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using WPF_Multi_Task.ViewModel;

namespace WPF_Multi_Task
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        MainViewModel ViewModel;
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            this.DataContext = ViewModel = new MainViewModel();
        }

        private void Window_Closed(object sender, EventArgs e)
        {
            ViewModel.SaveData();
        }
    }
}
```

MainWindow.xaml

:

```
<Window x:Class="WPF_Multi_Task.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WPF_Multi_Task"
        mc:Ignorable="d"
        Closed="Window_Closed"
        Loaded="Window_Loaded"
        Title="WPF Multi Task - K9 from Kteam" Height="450" Width="600">
    <Window.Resources>
        <Style TargetType="Button">
            <Setter Property="Margin" Value="5"></Setter>
            <Setter Property="Width" Value="75"></Setter>
            <Setter Property="Height" Value="25"></Setter>
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="Margin" Value="5"></Setter>
            <Setter Property="Width" Value="75"></Setter>
            <Setter Property="Height" Value="25"></Setter>
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>

        <StackPanel Orientation="Horizontal">
            <Button Content="Close All" Command="{Binding CloseAll_CMD}"></Button>
            <Button Content="Add data" Command="{Binding AddData_CMD}"></Button>
            <TextBox Text="{Binding SettingData.TotalData, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"></TextBox>
            <Button Content="Start all" Command="{Binding StartAll_CMD}"></Button>
            <Button Content="Stop all" Command="{Binding StopAll_CMD}"></Button>
            <Button Content="Delete all" Command="{Binding DeleteAll_CMD}"></Button>
        </StackPanel>

        <Grid Grid.Row="1">
            <ListView ItemsSource="{Binding Profiles}" Margin="5">
                <ListView.View>
                    <GridView>
                        <!--<GridViewColumn Header="STT" Width="40"
                                        DisplayMemberBinding="{Binding RelativeSource={RelativeSource FindAncestor,
                                         AncestorType={x:Type ListViewItem}},
                                         Converter={StaticResource IndexConverter}}" >
                        </GridViewColumn>-->
                        <GridViewColumn Header="STT" Width="30" DisplayMemberBinding="{Binding Index}"></GridViewColumn>
                        <GridViewColumn Header="Name" Width="100" DisplayMemberBinding="{Binding Name}">
                            <GridViewColumn.CellTemplate>
                                <DataTemplate>
                                    <TextBox Text="{Binding BindingGroupName, Mode=OneWay}" IsReadOnly="True"></TextBox>
                                </DataTemplate>
                            </GridViewColumn.CellTemplate>
                        </GridViewColumn>
                        <GridViewColumn Header="Status" Width="150" DisplayMemberBinding="{Binding Status}"></GridViewColumn>
                        <!--<GridViewColumn Header="User name" Width="150" DisplayMemberBinding="{Binding UserName}">
</GridViewColumn>

                        <GridViewColumn Header="Password" Width="150" DisplayMemberBinding="{Binding Password}"></GridViewColumn>--
>
                        <GridViewColumn >
                            <GridViewColumn.CellTemplate>
                                <DataTemplate>
                                    <Button Content="Open"
                                            Command="{Binding RelativeSource={RelativeSource FindAncestor,
                                         AncestorType={x:Type Window}},Path=DataContext.OpenDriver_CMD}"
```

```
                        CommandParameter="{Binding}"></Button>
                </DataTemplate>
            </GridViewColumn.CellTemplate>
        </GridViewColumn>
        <GridViewColumn >
            <GridViewColumn.CellTemplate>
                <DataTemplate>
                    <Button Content="Close"
                            Command="{Binding RelativeSource={RelativeSource FindAncestor,
                          AncestorType={x:Type Window}},Path=DataContext.CloseDriver_CMD}"
                            CommandParameter="{Binding}"></Button>
                </DataTemplate>
            </GridViewColumn.CellTemplate>
        </GridViewColumn>
        <GridViewColumn >
            <GridViewColumn.CellTemplate>
                <DataTemplate>
                    <Button Content="Stop"
                            Command="{Binding RelativeSource={RelativeSource FindAncestor,
                          AncestorType={x:Type Window}},Path=DataContext.StopProfile_CMD}"
                            CommandParameter="{Binding}"></Button>
                </DataTemplate>
            </GridViewColumn.CellTemplate>
        </GridViewColumn>
        <GridViewColumn >
            <GridViewColumn.CellTemplate>
                <DataTemplate>
                    <Button Content="Start"
                            Command="{Binding RelativeSource={RelativeSource FindAncestor,
                          AncestorType={x:Type Window}},Path=DataContext.StartProfile_CMD}"
                            CommandParameter="{Binding}"></Button>
                </DataTemplate>
            </GridViewColumn.CellTemplate>
        </GridViewColumn>
        <GridViewColumn >
            <GridViewColumn.CellTemplate>
                <DataTemplate>
                    <Button Content="Delete"
                            Command="{Binding RelativeSource={RelativeSource FindAncestor,
                          AncestorType={x:Type Window}},Path=DataContext.DeleteProfile_CMD}"
                            CommandParameter="{Binding}"></Button>
                </DataTemplate>
            </GridViewColumn.CellTemplate>
        </GridViewColumn>
        <GridViewColumn >
            <GridViewColumn.CellTemplate>
                <DataTemplate>
                    <Button Content="Pause"
                            Command="{Binding RelativeSource={RelativeSource FindAncestor,
                          AncestorType={x:Type Window}},Path=DataContext.PauseProfile_CMD}"
                            CommandParameter="{Binding}"></Button>
                </DataTemplate>
            </GridViewColumn.CellTemplate>
        </GridViewColumn>
        <GridViewColumn >
            <GridViewColumn.CellTemplate>
                <DataTemplate>
                    <Button Content="Resume"
                            Command="{Binding RelativeSource={RelativeSource FindAncestor,
                          AncestorType={x:Type Window}},Path=DataContext.ResumeProfile_CMD}"
                            CommandParameter="{Binding}"></Button>
                </DataTemplate>
            </GridViewColumn.CellTemplate>
        </GridViewColumn>
    </GridView>
</ListView.View>
```

```
            </ListView>
        </Grid>
    </Grid>
</Window>
```

Model\ProfileDetail.cs

:

```
using OpenQA.Selenium.Chrome;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using WPF_Multi_Task.ViewModel;

namespace WPF_Multi_Task.Model
{
    public class ProfileDetail : BaseViewModel
    {
        private int _Index;
        public int Index { get => _Index; set { _Index = value; OnPropertyChanged(); } }

        private string _Name;
        public string Name { get => _Name; set { _Name = value; OnPropertyChanged(); } }

        private string _Status;
        public string Status { get => _Status; set { _Status = value; OnPropertyChanged(); } }

        private string _UserName;
        public string UserName { get => _UserName; set { _UserName = value; OnPropertyChanged(); } }

        private string _Password;
        public string Password { get => _Password; set { _Password = value; OnPropertyChanged(); } }

        public ChromeDriver Driver { get; set; }
    }
}
```

Model\SettingData.cs

:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using WPF_Multi_Task.ViewModel;

namespace WPF_Multi_Task.Model
{
    public class SettingData : BaseViewModel
    {
        private int _TotalData;
        public int TotalData
        {
            get => _TotalData;
            set
            {

                _TotalData = value;
                OnPropertyChanged();
            }
        }
    }
}
```

ViewModel\BaseViewModel.cs

:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Input;

namespace WPF_Multi_Task.ViewModel
{
    public class BaseViewModel : ThreadController, INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
                handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
    public class ThreadController
    {
        public delegate void ThreadAction();
        public Task _Task;
        CancellationTokenSource src;
        PauseTokenSource pauseSource;
        public bool StopTask()
        {
            if (_Task == null)
                return true;
            try
            {
                if (src == null)
                    return true;
                src.Cancel();
                return true;
            }
            catch
            {
                return false;
            }
        }

        public void TaskWait(CancellationTokenSource TokenSrouce, PauseTokenSource PauseSource)
        {
            var ct = StartTask(async () =>
            {

                while (true)
                {
                    try
                    {
                        TokenSrouce.Token.ThrowIfCancellationRequested();
                    }
                    catch
                    {
                        return;
                    }
                    await PauseSource.Token.PauseIfRequestedAsync();

                    await Task.Delay(TimeSpan.FromSeconds(1));
                }
            }, TokenSrouce, PauseSource);
```

```
        }

        public bool StartTask(ThreadAction action, CancellationTokenSource TokenSrouce, PauseTokenSource PauseSource)
        {
            if (_Task != null)
            {
                StopTask();
                _Task = null;
            }
            try
            {
                src = TokenSrouce;
                pauseSource = PauseSource;

                if (TokenSrouce == null)
                {
                    _Task = Task.Run(() => { action(); });
                }
                else
                {
                    _Task = Task.Run(() => { action(); }, TokenSrouce.Token);
                }



                return true;
            }
            catch
            {
                return false;
            }
        }

        public async Task<bool> PauseTask()
        {
            if (_Task == null)
                return true;

            try
            {
                await pauseSource.PauseAsync();
                return true;
            }
            catch
            {
                return false;
            }
        }

        public async Task<bool> ResumeTask()
        {
            if (_Task == null)
                return true;

            try
            {
                await pauseSource.ResumeAsync();
                return true;
            }
            catch
            {
                return false;
            }
        }
    }
    public class PauseTokenSource
```

```
{
    bool _paused = false;
    bool _pauseRequested = false;

    TaskCompletionSource<bool> _resumeRequestTcs;
    TaskCompletionSource<bool> _pauseConfirmationTcs;

    readonly SemaphoreSlim _stateAsyncLock = new SemaphoreSlim(1);
    readonly SemaphoreSlim _pauseRequestAsyncLock = new SemaphoreSlim(1);

    public PauseToken Token { get { return new PauseToken(this); } }

    public async Task<bool> IsPaused(CancellationToken token = default(CancellationToken))
    {
        await _stateAsyncLock.WaitAsync(token);
        try
        {
            return _paused;
        }
        finally
        {
            _stateAsyncLock.Release();
        }
    }

    public async Task ResumeAsync(CancellationToken token = default(CancellationToken))
    {
        await _stateAsyncLock.WaitAsync(token);
        try
        {
            if (!_paused)
            {
                return;
            }

            await _pauseRequestAsyncLock.WaitAsync(token);
            try
            {
                var resumeRequestTcs = _resumeRequestTcs;
                _paused = false;
                _pauseRequested = false;
                _resumeRequestTcs = null;
                _pauseConfirmationTcs = null;
                resumeRequestTcs.TrySetResult(true);
            }
            finally
            {
                _pauseRequestAsyncLock.Release();
            }
        }
        finally
        {
            _stateAsyncLock.Release();
        }
    }

    public async Task PauseAsync(CancellationToken token = default(CancellationToken))
    {
        await _stateAsyncLock.WaitAsync(token);
        try
        {
            if (_paused)
            {
                return;
            }
```

```
            Task pauseConfirmationTask = null;

            await _pauseRequestAsyncLock.WaitAsync(token);
            try
            {
                _pauseRequested = true;
                _resumeRequestTcs = new TaskCompletionSource<bool>(TaskCreationOptions.RunContinuationsAsynchronously);
                _pauseConfirmationTcs = new TaskCompletionSource<bool>(TaskCreationOptions.RunContinuationsAsynchronously);
                pauseConfirmationTask = WaitForPauseConfirmationAsync(token);
            }
            finally
            {
                _pauseRequestAsyncLock.Release();
            }

            await pauseConfirmationTask;

            _paused = true;
        }
        finally
        {
            _stateAsyncLock.Release();
        }
    }

    private async Task WaitForResumeRequestAsync(CancellationToken token)
    {
        using (token.Register(() => _resumeRequestTcs.TrySetCanceled(), useSynchronizationContext: false))
        {
            await _resumeRequestTcs.Task;
        }
    }

    private async Task WaitForPauseConfirmationAsync(CancellationToken token)
    {
        using (token.Register(() => _pauseConfirmationTcs.TrySetCanceled(), useSynchronizationContext: false))
        {
            await _pauseConfirmationTcs.Task;
        }
    }

    internal async Task PauseIfRequestedAsync(CancellationToken token = default(CancellationToken))
    {
        Task resumeRequestTask = null;

        await _pauseRequestAsyncLock.WaitAsync(token);
        try
        {
            if (!_pauseRequested)
            {
                return;
            }
            resumeRequestTask = WaitForResumeRequestAsync(token);
            _pauseConfirmationTcs.TrySetResult(true);
        }
        finally
        {
            _pauseRequestAsyncLock.Release();
        }

        await resumeRequestTask;
    }
}

// PauseToken - consumer side
public struct PauseToken
```

```
    {
        readonly PauseTokenSource _source;

        public PauseToken(PauseTokenSource source) { _source = source; }

        public Task<bool> IsPaused() { return _source.IsPaused(); }

        public Task PauseIfRequestedAsync(CancellationToken token = default(CancellationToken))
        {
            return _source.PauseIfRequestedAsync(token);
        }
    }
    class RelayCommand<T> : ICommand
    {
        private readonly Predicate<T> _canExecute;
        private readonly Action<T> _execute;

        public RelayCommand(Predicate<T> canExecute, Action<T> execute)
        {
            if (execute == null)
                throw new ArgumentNullException("execute");
            _canExecute = canExecute;
            _execute = execute;
        }

        public bool CanExecute(object parameter)
        {
            return _canExecute == null ? true : _canExecute((T)parameter);
        }

        public void Execute(object parameter)
        {
            _execute((T)parameter);
        }

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }
    }
}
```

ViewModel\MainViewModel.cs

:

```
using Newtonsoft.Json;
using OpenQA.Selenium.Chrome;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using WPF_Multi_Task.Model;

namespace WPF_Multi_Task.ViewModel
{
    public class MainViewModel : BaseViewModel
    {
        #region Properties
        private SettingData _SettingData;
        public SettingData SettingData { get => _SettingData; set { _SettingData = value; OnPropertyChanged(); } }

        private ObservableCollection<ProfileDetail> _Profiles;
        public ObservableCollection<ProfileDetail> Profiles
        {
            get => _Profiles;
            set
            {

                _Profiles = value;
                OnPropertyChanged();
            }
        }

        #endregion

        public MainViewModel()
        {
            FirstLoad();
            LoadCommand();
        }

        #region CMD
        public ICommand AddData_CMD { get; set; }
        public ICommand PauseProfile_CMD { get; set; }
        public ICommand ResumeProfile_CMD { get; set; }
        public ICommand StartProfile_CMD { get; set; }
        public ICommand DeleteProfile_CMD { get; set; }
        public ICommand CreateProfile_CMD { get; set; }
        public ICommand StartAll_CMD { get; set; }
        public ICommand StopAll_CMD { get; set; }
        public ICommand StopProfile_CMD { get; set; }
        public ICommand DeleteAll_CMD { get; set; }
        public ICommand CloseAll_CMD { get; set; }
        public ICommand OpenDriver_CMD { get; set; }
        public ICommand CloseDriver_CMD { get; set; }
        #endregion

        #region Method
        void FirstLoad()
        {
            LoadSavedData();
        }
```

```
        void LoadCommand()
        {
            StopProfile_CMD = new RelayCommand<ProfileDetail>((p) => { return p != null && Profiles != null &&
Profiles.Contains(p); }, (p) => { StopProfile(p); });
            AddData_CMD = new RelayCommand<ProfileDetail>((p) => { return true; }, (p) => { Add500Row(); });
            PauseProfile_CMD = new RelayCommand<ProfileDetail>((p) => { return p != null && Profiles != null &&
Profiles.Contains(p); }, (p) => { PauseProfile(p); });
            ResumeProfile_CMD = new RelayCommand<ProfileDetail>((p) => { return p != null && Profiles != null &&
Profiles.Contains(p); }, (p) => { ResumeProfile(p); });
            StartProfile_CMD = new RelayCommand<ProfileDetail>((p) => { return p != null && Profiles != null &&
Profiles.Contains(p); }, (p) => { StartProfile(p); });
            DeleteProfile_CMD = new RelayCommand<ProfileDetail>((p) => { return p != null && Profiles != null &&
Profiles.Contains(p); }, async (p) =>{ StopProfile(p); await DeleteProfile(p);});
            CreateProfile_CMD = new RelayCommand<object>((p) => { return true; }, (p) => { CreateProfile(); });
            StartAll_CMD = new RelayCommand<ProfileDetail>((p) => { return Profiles != null; }, (p) => { StartAll(); });
            StopAll_CMD = new RelayCommand<ProfileDetail>((p) => { return Profiles != null; }, (p) => { StopAll(); });
            DeleteAll_CMD = new RelayCommand<ProfileDetail>((p) => { return Profiles != null; }, (p) => { DeleteAll(); });
            CloseAll_CMD = new RelayCommand<ProfileDetail>((p) => { return true; }, (p) => { CloseAllDriver(); });
            OpenDriver_CMD = new RelayCommand<ProfileDetail>((p) => { return p != null; }, (p) => { p.StartTask(() => {
OpenDriver(p); }, null,null);  });
            CloseDriver_CMD = new RelayCommand<ProfileDetail>((p) => { return p != null; }, (p) => { p.StartTask(() => {
CloseDriver(p); }, null, null);  });
        }
        void Demo()
        {
            MessageBox.Show("DDDDD");
        }

        void StopProfile(ProfileDetail p)
        {
            p.Status = "Being stop";
            p.StopTask();
            p.Status = "Stop";
        }

        async void PauseProfile(ProfileDetail p)
        {
            if (await p.PauseTask())
            {
                p.Status = "Paused";
            }
        }

        async void ResumeProfile(ProfileDetail p)
        {
            if (await p.ResumeTask())
            {
                p.Status = "Resumed";
            }
        }

        void StartAll()
        {
            StartTask(() => {
                foreach (var item in Profiles)
                {
                    StartProfile(item);
                }
            }, null, null);
        }

        void CloseAllDriver()
        {
            StartTask(() => {
                foreach (var item in Profiles)
                {
```

```
                StopProfile(item);
                CloseDriver(item);
            }
        }, null, null);
    }
    void DeleteAll()
    {
        StartTask(async() => {
            while (Profiles.Count > 0)
            {
                var item = Profiles.First();
                StopProfile(item);
                await DeleteProfile(item);
            }

        }, null, null);
    }

    void StopAll()
    {
        StartTask(() => {
            foreach (var item in Profiles)
            {
                StopProfile(item);
            }
        }, null, null);
    }

    void Add500Row()
    {
        StartTask(() => {
            for (int i = 0; i < SettingData.TotalData; i++)
            {
                CreateProfile(i + "");
            }
        }, null, null);
    }

    void OpenDriver(ProfileDetail p)
    {
        CloseDriver(p);
        p.Driver = new ChromeDriver();
    }

    void CloseDriver(ProfileDetail p)
    {
        if (p.Driver != null)
        {
            try
            {
                p.Driver.Quit();
            }
            catch
            {

            }
        }
    }

    void StartProfile(ProfileDetail p)
    {
        CancellationTokenSource sourceToken = new CancellationTokenSource();
        PauseTokenSource pauseToken = new PauseTokenSource();
        var ct = p.StartTask(async () =>
        {
            OpenDriver(p);
```

```
            p.Driver.Navigate().GoToUrl("https://howkteam.vn/");

            while (true)
            {
                p.Status = $"Working...{DateTime.Now.Second}";
                p.Driver.Navigate().GoToUrl("https://howkteam.vn/learn");
                await Task.Delay(TimeSpan.FromSeconds(5));
                try
                {
                    sourceToken.Token.ThrowIfCancellationRequested();
                }
                catch
                {
                    return;
                }
                await pauseToken.Token.PauseIfRequestedAsync();


                p.Driver.Navigate().GoToUrl("https://howkteam.vn/question");
                await Task.Delay(TimeSpan.FromSeconds(5));
                try
                {
                    sourceToken.Token.ThrowIfCancellationRequested();
                }
                catch
                {
                    return;
                }
                await pauseToken.Token.PauseIfRequestedAsync();
            }

        }, sourceToken, pauseToken);
    }

    async Task DeleteProfile(ProfileDetail p)
    {
        p.Status = $"Deleting Profile";
        await Application.Current.Dispatcher.InvokeAsync(new Action(() =>
        {
            p.Status = $"Removing";
            Profiles.Remove(p);
            UpdateProfileIndex();
        }));
    }

    void UpdateProfileIndex()
    {
        if (Profiles == null || Profiles.Count == 0)
        {
            return;
        }
        int i = 0;
        foreach (var item in Profiles)
        {
            item.Index = ++i;
        }
    }

    ProfileDetail CreateProfile(string profileName = null)
    {
        if(Profiles == null)
        {
            Profiles = new ObservableCollection<ProfileDetail>();
        }
        var profile = new ProfileDetail() { Name = profileName, Status = "Created" };
        Application.Current.Dispatcher.InvokeAsync(new Action(() =>
```

```
            {
                Profiles.Add(profile);
                UpdateProfileIndex();
            }));

            return profile;
        }

        void LoadSavedData()
        {
            try
            {
                var text = File.ReadAllText("Saved.txt");
                SettingData = JsonConvert.DeserializeObject<SettingData>(text);

            }
            catch
            {

            }

            if (SettingData == null)
            {
                SettingData = new SettingData();
                SettingData.TotalData = 100;
            }
        }

        public void SaveData()
        {
            try
            {
                File.WriteAllText("Saved.txt", JsonConvert.SerializeObject(SettingData));
            }
            catch { }
        }
        #endregion
    }
}
```

Kết luận

Qua bài học này, bạn đã nắm được Cách sử dụng Multi Task và đã biết được những kinh nghiệm để tùy chỉnh nó vào dự án cá nhân.

Cảm ơn các bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thử thách – Không ngại khó**".