Background Tasks

# Der BackgroundService / IBackgroundService

public interface IBackgroundService

{

/// <summary>

/// Gets the registered Instance of the requested Task type

/// </summary>

/// <typeparam name="T">Task Type</typeparam>

/// <returns>Registered Instance of T</returns>

T getTaskObject<T>() where T : BackgroundTask;

/// <summary>

/// Starts an invocation of the given Background Task or queues it, if the Task is busy

/// </summary>

/// <typeparam name="T">Task Type</typeparam>

/// <param name="argument">Argument for the Task ( Task specific )</param>

void startTask<T>(object argument) where T : BackgroundTask;

}

Ist für die Verwaltung von Background Tasks zuständig.

Erreichbar aus dem IOC Container mit: IOC.Resolve<IBackgroundService>()

Oder: App.BackgroundTasks

# Registrieren von Tasks

Neue Tasks müssen mit dem Service registriert werden. Dies geschieht im App Konstruktor, bei der Initialisierung.

backg.registerTask(new DownloadTaxonListTask(IOC));

Von jedem Task Typ existiert nur eine Instanz, die auch nur einen Aufruf gleichzeitig verarbeiten kann.

# Starten von Tasks

Gegeben eine Referenz auf den BackgroundService „background“ kann ein Task wie folgt gestartet werden:

background.startTask<TaskType>(argument);

# Das IBackgroundTask Interface

public interface IBackgroundTask

{

/// <summary>

/// Sends the argument object used to invoke the Task on completion

/// </summary>

IObservable<object> AsyncCompletedNotification { get; }

/// <summary>

/// Sends the argument object used to invoke the Task on cleanup

/// </summary>

IObservable<object> AsyncCleanupNotification { get; }

/// <summary>

/// Sends the argument object used to invoke the Task on execution

/// </summary>

IObservable<object> AsyncStartedNotification { get; }

/// <summary>

/// Sends task provided string status messages during execution

/// </summary>

IObservable<string> AsyncProgressMessages { get; }

/// <summary>

/// Sends either 0s or 1s depending on whether this task is currently executing or not

/// </summary>

IObservable<int> ItemsInflight { get; }

/// <summary>

/// Contains either 0 or 1 depending on whether this task is currently executing or not

/// </summary>

int CurrentItemsInFlight { get; }

/// <summary>

/// Contains the argument object used by the current task invocation

/// </summary>

object CurrentArguments { get; }

}

Bietet Informationen über den Ausführungszustand des aktuellen Tasks.

# Implementieren von Tasks

Alle Background Tasks müssen von der BackgroundTask Basisklasse ableiten.  
Diese bietet eine Reihe von Hilfsfunktionen für die ableitenden Klassen.

public abstract class BackgroundTask : IBackgroundTask

{

#region subclass Interface

/// <summary>

/// The State Dictionary for this Invocation that can be used to store

/// - The Argument of this Invocation

/// - The partial results of this Invocation for the purpose of resuming it

/// - etc.

/// </summary>

protected Dictionary<string, string> State { get; }

/// <summary>

/// This Flag is set, after the Cancel Method has run.

/// The Task is expected to refrain from changing the state Dictionary after this flag is set.

/// </summary>

public bool Cancelled { get; }

/// <summary>

/// Reports a given status string via the AsyncProgressMessages Property

/// </summary>

/// <param name="message">status string</param>

protected void reportProgress(string message);

/// <summary>

/// Indicates whether this type of Task can resume.

/// Tasks that cannot resume are cleaned up before reinvocation

/// those that can are expected to handle resuming on their own

/// </summary>

public abstract bool CanResume { get; }

/// <summary>

/// Saves the argument Object to the State Dictionary, so that it can be restored later.

/// </summary>

/// <param name="arg">Invocation Argument</param>

protected abstract void saveArgumentToState(object arg);

/// <summary>

/// Restores the argument Object from the State Dictionary, after it has been stored there

/// </summary>

/// <returns>Invocation Argument</returns>

protected abstract object getArgumentFromState();

/// <summary>

/// Task Entry Point for the executing background Thread

/// </summary>

protected abstract void Run(object arg);

/// <summary>

/// Cancels the Task

/// When this method returns, the cancellation must have been processed

/// </summary>

protected abstract void Cancel();

/// <summary>

/// Cleans up the effects of partial invocation of the task

/// After this has run, it must be safe to restart the invocation.

/// </summary>

/// <param name="arg">Invocation Argument</param>

protected abstract void Cleanup(object arg);

#endregion

# Ausführen von Tasks

1. background.startTask<TaskType>(argument);
   1. Task TaskType ist beschäftigt -> Aufruf wird in die Warteschlange eingereiht
   2. Task TaskType ist frei -> 2.
2. Task.saveArgumentToState(argument) wird aufgerufen.
3. Task.Run(argument) wird auf dem Arbeiter Thread aufgerufen
4. Task wird unterbrochen.
   1. Die Anwendung wird wieder fortgesetzt (Tombstoning – Reactivation)  
      🡪 Der Task läuft von selbst weiter. Exceptions, die dabei auftreten muss der Task intern abfangen. (z.B. WebException bei WCF Service Aufrufen)
   2. Die Anwendung wird neu gestartet
      1. Task.getArgumentFromState wird aufgerufen, um das Argument wiederherzustellen.
      2. Der Task kann nicht fortsetzen (CanResume == false) ?  
         🡪 Task.Cleanup(argument) wird aufgerufen
      3. Goto 3.