**HRM CODE TUTORIAL COLLECTION**

**Collection By:** R&D Team

Index

[REQUEST POST LARGE DATA JSON – WEB MVC 5](#_Toc23858538)

[Features 5](#_Toc23858539)

[Resolve Exception 5](#_Toc23858540)

[Access to the Code 5](#_Toc23858541)

[How to use 5](#_Toc23858542)

[Example 6](#_Toc23858543)

[Access to the Code 19](#_Toc23858558)

[How to use 19](#_Toc23858559)

[Example 20](#_Toc23858560)

# 

# Quartznet

## Features

Quartz.NET là một full-featured open source job scheduling system, có thể được sử dụng từ những ứng dúng đơn giản cho tới quy mô hệ thống doanh nghiệp

Quartz.NET bao gồm hai khái niệm chính:

* **Job**: là những background task mà chúng ta muốn chạy theo một lịch trình cụ thể.
* **Scheduler**: chịu trách nhiệm chạy các Jobs dựa trên trigger, dựa trên lịch được lên sẵn trước đó.

ASP.NET Core hỗ trợ mạnh mẽ việc chạy các Background task thông qua **hosted services. Hosted Service** được chạy khi ASP.NET App bắt đầu chạy và quá trình chạy nền diễn ra trong suốt vòng đời của Application. Bằng việc tạo ra một Quartz.NET hosted serive, chúng ta có thế sử dụng standard ASP.NET Core Application để chạy các Task dưới background.

Trong khi chúng ta có thể tạo một “time” background service (Ví dụ một task sẽ được chạy sau mỗi 10 phút). Quartz.NET cung cấp một giải pháp vững chắc hơn nhiều. Chúng ta có thể tạo ra những Tasks chạy vào những thời điểm cụ thể khác nhau trong ngày, hoặc chỉ chạy trong những ngày cụ thể, hoặc hơn nữa là kết hợp bất kỳ lịch trình nào thông qua việc sử dụng **Cron trigger**. Quartz cũng cho phép chúng ta chạy nhiều **instances** của application trên một **Clustered fahsion,** từ đó duy nhất một **instance** có thể chạy một Task được chỉ định tại một thời điểm.

## Update Note

* 05/11/2019: Release

## Access to the Code

##### Server Side

Function **vnr\_validateDateFrom\_DateTo** *in* HRM.Presentation.Main → js → vnr\_Hrm\_Function.js

## How to use

### Installing Quartz.NET

Quartz.NET là một .NET Standard 2.0 Nuget package, vì vậy sẽ rất đơn giản để có thể cài đặt trong ứng dụng của chúng ta.

Cài đặt Quartz.NET thông qua lệnh dotnet add package Quartz, hoặc thông qua Manage Nuget Packages.

Bắt đầu sử dụng Quartz.NET

Ví dụ đơn giản tổng thể về Quartz.NET

// construct a scheduler factory

StdSchedulerFactory factory = new StdSchedulerFactory();

// get a scheduler

IScheduler scheduler = await factory.GetScheduler();

await scheduler.Start();

// define the job and tie it to our HelloJob class

IJobDetail job = JobBuilder.Create<HelloJob>()

.WithIdentity("myJob", "group1")

.Build();

// Trigger the job to run now, and then every 40 seconds

ITrigger trigger = TriggerBuilder.Create()

.WithIdentity("myTrigger", "group1")

.StartNow()

.WithSimpleSchedule(x => x

.WithIntervalInSeconds(40)

.RepeatForever())

.Build();

await scheduler.ScheduleJob(job, trigger);

// You could also schedule multiple triggers for the same job with

// await scheduler.ScheduleJob(job, new List<ITrigger>() { trigger1, trigger2 }, replace: true);

Trước khi có thể sử dụng scheduler, chúng ta cần khởi tạo một StdSchedulerFactory factory = new StdSchedulerFactory().

### Job và trigger

#### Key interface và class của Quartz API

* IScheduler: API chính tương tác với scheduler.
* IJob: interface cần được implement bởi component chúng ta muốn thực thi bằng scheduler.
* IJobDetail: define một instance of Jobs.
* ITrigger: một component dùng để define lịch trình cho một Job được chỉ định thực thi, job có thể có nhiều Trigger liên kết tới.
* JobBuiler: dùng để define/build JobDetail instances.
* TriggerBuilder: dùng để define/build Trigger instances.
* ScheduleBuilder: dùng để define/build Scheduler instances, yêu cầu Quartz.NET 3.1 hoặc mới hơn.

#### Vòng đời của một scheduler

Một scheduler có vòng đời gắn với việc khởi tạo chúng thông qua SchedulerFactory và gọi Shutdown() method. Một khi được tạo IScheduler Interface có thể được sử dụng để thêm, xóa và liệt kê danh sách Jobs và Triggers, hoặc thực thi một số thao tác liên quan (ví dụ như tạm dừng một trigger). Tuy nhiên một Scheduler sẽ không thực thi bất kỳ một Trigger nào cho tới khi nó được khởi động (thông qua Start() method)

#### Builder classes

Quartz cung cấp “builder” classes định nghĩa một Domain Specific Language (DSL, đôi khi gọi là “fluent interface”)

// define the job and tie it to our HelloJob class

IJobDetail job = JobBuilder.Create<HelloJob>()

.WithIdentity("myJob", "group1") // name "myJob", group "group1"

.Build();

// Trigger the job to run now, and then every 40 seconds

ITrigger trigger = TriggerBuilder.Create()

.WithIdentity("myTrigger", "group1")

.StartNow()

.WithSimpleSchedule(x => x

.WithIntervalInSeconds(40)

.RepeatForever())

.Build();

// Tell quartz to schedule the job using our trigger

await sched.scheduleJob(job, trigger);

Block code phía trên sử dụng JobBulder để build một Job thông qua cách sử dụng Fluent Interface

#### Jobs and Triggers

##### Job

Một Job là một class implement IJob interface (có duy nhất một method đơn giản là Execute())

IJob interface

namespace Quartz

{

public interface IJob

{

Task Execute(JobExecutionContext context);

}

}

Một khi Job’s trigger được kích hoạt, Execute() method sẽ được gọi bởi Scheduler’s Worker Thread, JobExecutionContext object sẽ cung cấp các thông tin liên quan đến Job instance và môi trường “run-time” của nó.

JobDetail object được khởi tạo bởi Quartz.NET client (app) tại thời điểm Job được thêm vào Scheduler. Nó chứa các Property Settings cho Job cùng với JobDataMap được sử dụng để lưu trữ trạng thái thông tin được cung cấp cho Job instance (class kế thừa từ IJob).

##### Trigger

Trigger object được sử dụng để thực thi một Jobs. Khi chúng ta mong muốn lên lịch cho một Job, chúng ta khởi tạo một trigger và cung cấp lịch chúng ta muốn thực thi. Trigger cũng có một JobDataMap liên kết, việc này sẽ có ích cho việc passing parameters tới một Job được chỉ định thực thi bởi trigger.

Quartz cung cấp một bộ Trigger types khác nhau, các Trigger types được sử dụng nhiều nhất là SimpleTrigger (intertface ISimpleTrigger) và CronTrigger (interface ICronTrigger).

SimpleTrigger tiện lợi khi chúng ta cần thực thi một Job tại một thời điểm được chỉ định trước, hoặc lặp lại trong N lần với một khoảng thời gian T delay giữa các lần thực thi.

CronTrigger hữu dụng nếu chúng ta muốn sắp lịch thực thi cụ thể ví dụ như “thực thi mỗi buổi tối thứ Năm hàng tuần”, hoặc “10:15 ngày thứ 10 hàng tháng”.

##### Jobs và Triggers

Nhiều trường hợp Job Scheduler không có một khái niệm cụ thể về Jobs và Triggers. Một số định nghĩa một ‘Job’ đơn giản là một công việc được thực thi một lần với một số lượng nhỏ Job Identifier. Số khác có thể ketes hợp Job và Trigger objects

Quartz cung cấp việc tạo lịch trình và công việc thực thi tách biệt. Ví dụ như Job có thể được tạo và lưu trữ tại Job scheduler độc lập với Trigger, và nhiều Triggers có thể liên kết tới cùng một Job

### Identities

Job và Trigger được cung cấp identity keys khi register Quartz Scheduler. Key của Job và Trigger (JobKey và TriggerKey) cho phép chúng được đặt vào các Groups, việc này có ích khi chúng ta muốn tổ chức Jobs và Triggers vào các Categories như “Reporting Jobs” hoặc “Maintenance Jobs”. Key name của Job và Trigger phải là unique bên trong một Group.

Nói cách khác, một key (identifier) hoàn chỉnh của một Job hoặc Trigger được đi kèm với tên của Group.

### Chi tiết hơn về Jobs và JobDetails

Như thông tin về Job ở trên, Jobs rất đơn giản để có thể implement. Tuy nhiên chúng ta cần phải hiểu thêm một vài thông tin về Job, về Execute(…) method của IJob interface và JobDetails.

Trong khi Job class (implement IJob interface) được xử dụng để thực thi công việc chúng ta mong muốn. Quartz.NET cần được cung cấp thông tin mà chúng ta muốn Job Instance xử lý. Việc này được thực hiện thông qua JobDetails class

JobDetails Instances được build sử dụng JobBuilder class. JobBuilder cho phép chúng ta mô tả Job thông qua Fluent Interface

#### JobDataMap

JobDataMap có thể được sử dụng để giữ các Serializeble Objects mà chúng ta muốn chúng được cung cấp cho Job Instances khi chúng được thực thi. JobDataMap implement IDictionary interface, và có thêm các methods hữu ích cho việc Sorting và Retrieving data của các primitive types.

Setting Values trong một JobDataMap

// define the job and tie it to our DumbJob class

IJobDetail job = JobBuilder.Create<DumbJob>()

.WithIdentity("myJob", "group1") // name "myJob", group "group1"

.UsingJobData("jobSays", "Hello World!")

.UsingJobData("myFloatValue", 3.141f)

.Build();

Getting Values từ JobDataMap

public class DumbJob : IJob

{

public async Task Execute(IJobExecutionContext context)

{

JobKey key = context.JobDetail.Key;

JobDataMap dataMap = context.JobDetail.JobDataMap;

string jobSays = dataMap.GetString("jobSays");

float myFloatValue = dataMap.GetFloat("myFloatValue");

await Console.Error.WriteLineAsync("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " + myFloatValue);

}

}

Trường hợp nếu chúng ta sử dụng Persistent JobStore, nên cân nhắc về việc quyết định đưa những gì vào JobDataMap, vì các Object đặt trong đó sẽ được Serialized, và từ đó dẫn tới vấn đề class-versioning.

Hiển nhiên, đối với Standard .NET types được sử dụng tốt, tuy nhiên bất kỳ thời điểm nào một ai đó thay đổi defination của một class từ đó chúng ta có serialized instances, trường hợp này cần cẩn thận để không phá vỡ tính tương thích.

Một lựa chọn khác, chúng ta có thể đặt AdoJobStore và JobDataMap vào một mode nơi chỉ có những primitive types và strings có thể được lưu trữ trong map, từ đó loại bỏ bất kỳ vấn đề serialization nào phát sinh trong tương lai.

Nếu chungsta thêm Properties với Set Accessor cho Job Class tương ứng với tên của Key JobDataMap, Quartz default JobFactory implementation sẽ tự động gọi những setters đó khi Job được khởi tạo. Từ đó loại bỏ việc phải tường minh get Values từ Map trong Execute() method. Note, chức năng này không được mặc định maintain khi sử dụng JobFactory.

Trigger cũng có thể được liên kết với JobDataMaps. Điều này có ý nghĩa trong trường hợp chúng ta có một Job nào đó được lưu trữ trong Scheduler được tác động bởi nhiều Triggers, vì các Trigger này là các Trigger độc lập với nhau, nên trong trường hợp này chúng ta có thể sẽ cần các Data Inputs khác nhau dựa vào Trigger JobDataMaps.

JobDataMap nằm trong JobExecutionContext trong khi một Job đang trong quá trình thực thi. Nó được kết hợp với JobDataMap trong JobDetail và trong Trigger, (with values in the later overriding any same-named values in the former). …….

…

….

#### Job “Instances”

Job Instance là một khái niệm có thể khiến nhiều người tốn một lượng lớn thời gian để hiểu chính xác những gì cấu thành chúng. Khái niệm này cần được định nghĩa cụ thể tại mục này và “Job State and Concurrency”.

Về cơ bản, chúng ta có thể tạo ra một Job Class, và lưu trữ nhiều “Instance Definiations” của chúng trong Scheduler bằng cách tạo ra nhiều Instances của JobDetails.

* Mỗi trong số những Instances đó sẽ có Properties và JobDataMap riêng biệt, và thêm tất cả chúng vào Scheduler.

Ví dụ, chúng ta tạo một Class implements IJob Interface gọi là “SalesReportJob”. Và SalesReportJob Class sẽ được thực thi thông qua parameters chúng ta truyền vào (thông qua JobDataMap) để chỉ định cụ thể tên người thực hiện Job đó (ví dụ PersonA bán cho PersonB trong Job Instance 1). Và trong nhiều trường hợp chúng ta sẽ tạo ra Multiple Definitions (JobDetails) của một Job, JobMapData sẽ thực hiện chức năng Input Data cho từng Job một cách cụ thể.

Khi một Trigger được kích hoạt, JobDetails (Instance Definition) được liên kết tới sẽ được load, và Job class được Refer tới sẽ được khởi tạo thông qua JobFactory cấu hình tại Scheduler. Default JobFactory sẽ gọi Default Constructor của Job Class sử dụng Activator.CreateInstance, sau đó tìm kiếm và gọi Setter Properties trong Job Class trùng với tên của Keys trong JobDataMap.

#### Job State and Concurrency

Thêm một vài lưu ý về Job’s State Data (aka JobDataMap) và Concurrency. Có một vài thuộc tính (Attribute) có thể được thêm vào Job Class sẽ ảnh hưởng tới Quartz’s Behaviour.

[DisallowConcurrentExecution] là một Attribute có thể được thêm vào Job Class chỉ định Quartz không thực thi Multiple Instances của Job Class một cách đồng thời. *Notes, nếu “SalesReportJob” có Attribute này, việc này nghĩa là duy nhất một Instance của “SalesReportForPersonA” có thể được thực thi tại một thời điểm, nhưng vẫn có thể thực thi đồng thời một Instance của “SaleReportForPersonB”.*

[PersistJobDataAfterExecution] có thể thêm vào Job Class chỉ định Quartz Update bản sao của “JobDetail’s JobDataMap” sau khi Execute() method hoàn tất thành công (không throw exception), như vậy việc thực thi một Job tương tự tiếp theo (JobDetail) nhận được Updated Values thay vì giá trị ban đầu.

Nếu chúng ta sử dụng [PersistJobDataAfterExecution] Attribute, cần phải cân nhắc việc sử dụng [DisallowConcurrentExecution] Attribute để tránh xảy ra việc không phân biệt được Data nào đã bị loại khỏi Store khi 2 Instances của cùng một Job (JobDetail) thực thi đồng thời.

### Chi tiết về Triggers

Bên cạnh việc tất cả mọi Trigger Types đều có TriggerKey Property để theo dõi định danh của chúng, ngoài ra còn có một số Properties khác mà tất cả Trigger Types cũng nắm giữ.

Danh sách các Properties phổ biến với tất cả Trigger Types

#### Danh sách các Properties phổ biến với tất cả Trigger Types

* JobKey Property là định danh của Job cần được thực thi khi Trigger kích hoạt
* StartTimeUtc Property chỉ định thời gian khi nào Trigger’s Schedule lần đầu tiên được hoạt động. Giá trị của StartTimeUtc là một DateTimeOffset Object. Đối với một vài Trigger Types, Trigger sẽ được kích hoạt tại thời điểm ban đầu, một số khác sẽ được đánh dấu một mốc thời gian trên Schedule về thời điểm hoạt động. Điều này có ý nghĩa chúng ta có thể lưu trữ một Trigger với một Schedule như “Tại mỗi ngày thứ 5 của tháng”, hiện tại đang là tháng 1, và nếu StartTimeUtc Property được set là ngày 1-4, nó sẽ cần khoảng thời gian vài tháng trước khi lần đầu tiên Trigger được kích hoạt.
* EndTimeUtc Property chỉ định Trigger’s Schedule không còn được hoạt động nữa. Nói cách khách, nếu một Trigger với một Schedule “mỗi ngày thứ 5 của tháng” và EndTimeUtc được set là ngày 1-7 thì sau ngày đó Trigger sẽ không hoạt động nữa.

#### Priority

Đôi khi chúng ta có rất nhiều Trigger (hoặc một vài Worker Thread trong Quartz.NET Thread Pool), Quartz.NET có thể sẽ không đủ tài nguyên để ngay lập tức kích hoạt toàn bộ Trigger mà chúng ta đã lên lịch tại cùng một thời điểm. Trong trường hợp này, chúng ta có thể sẽ cần kiểm soát việc Trigger được kích hoạt theo một thứ tự ưu tiên được xác định. Để thực hiện mục đích này, chúng ta có thể set Priority Property cho Trigger. Nếu N Trigger được kích hoạt tại cùng một thời điểm, và chỉ có duy nhất Z Worker Threads hiện tại đang khả dụng, từ đó Z Triggers đầu tiên với độ ưu tiên cao nhất sẽ được kích hoạt trước. Nêu chúng ta không set Priority cho Trigger, Trigger sẽ có Default Priority là 5. Trigger Priority Value là bất kỳ một số nguyên nào, cả âm và dương.

#### Misfire Instructions

Một Property quan trọng khác của Trigger là “Misfire Instruction”. Một Misfire xảy ra nếu một Persistent Trigger “bị lỡ” việc kích hoạt, nguyên nhân có thể do Scheduler bị Shutdown, hoặc không có Thread nào khả dụng trong Quartz.NET Thread Pool để thực thi Job. Với mỗi Trigger Types sẽ có Misfire Instruction riêng cho chúng. Default Misfire Instruction của các Trigger là “Smart Policy” Instruction, nó có một Dynamic Behavior dựa trên Trigger Type và Configuration. Khi Scheduler hoạt động, nó sẽ tìm kiếm bất kỳ Persistent Trigger nào đã bị “lỡ” kích hoạt, và nó sẽ Update các Trigger đó dựa trên cấu hình Misfire Instruction của từng Trigger riêng biệt. Khi bắt đầu sử dụng Quartz.NET chúng ta cần làm quen với việc sử dụng Misfire Instruction được định nghĩa cho các Trigger Types khác nhau.

#### Calendars

Quartz.NET Calendar Object implement Icalendar Interface có thể liên kết với các Triggers tại thời điểm Trigger được lưu trữ trong Scheduler. Calendar hữu ích trong việc làm một số Trigger không cần thực hiện trong Schedule. Ví dụ chúng ta tạo ra một Trigger để kích hoạt một Job trong mỗi 9:30 sáng mọi ngày trong tuần, tuy nhiên trong các ngày nghỉ Lễ chúng ta không muốn Trigger này hoạt động, việc này có thể được thực hiện bằng việc thêm một Calendar loại bỏ các ngày nghỉ Lễ đó.

Calendar có thể là bất kỳ Serializable Object nào implement ICalendar Interface

namespace Quartz

{

public interface ICalendar

{

string Description { get; set; }

ICalendar CalendarBase { set; get; }

bool IsTimeIncluded(DateTimeOffset timeUtc);

DateTime GetNextIncludedTimeUtc(DateTimeOffset timeUtc);

ICalendar Clone();

}

}

Mặc dù Calendars có thể loại bỏ các khoảng thời gian nhỏ tới mức milisecond, tuy nhiên chúng ta thường mong muốn việc loại bỏ cả ngày hơn. Và trong Quartz.NET cung cấp một class HolidayCalendar thuận tiện cho việc này.

Calendars phải được khởi tạo (Instantiate) và đăng ký (Register) với Scheduler thông qua AddCalendar(…) method. Nếu chúng ta sử dụng HolidayCalendar, sau khi khởi tạo chúng, chúng ta cần sử dụng AddExcludedDate(DateTime date) method để sử dụng

Ví dụ

HolidayCalendar cal = new HolidayCalendar();

cal.AddExcludedDate(someDate);

await sched.AddCalendar("myHolidays", cal, false);

ITrigger t = TriggerBuilder.Create()

.WithIdentity("myTrigger")

.ForJob("myJob")

.WithSchedule(CronScheduleBuilder.DailyAtHourAndMinute(9, 30)) // execute job daily at 9:30

.ModifiedByCalendar("myHolidays") // but not on holidays

.Build();

// .. schedule job with trigger

ITrigger t2 = TriggerBuilder.Create()

.WithIdentity("myTrigger2")

.ForJob("myJob2")

.WithSchedule(CronScheduleBuilder.DailyAtHourAndMinute(11, 30)) // execute job daily at 11:30

.ModifiedByCalendar("myHolidays") // but not on holidays

.Build();

// .. schedule job with trigger2

### SimpleTrigger

Simple Trigger đáp ứng được nhu cầu Schedule của chúng ta trong trường hợp chúng ta muốn một Job thực hiện vào chính xác một thời điểm nào đó, lặp lại bao nhiêu lần sau một khoảng thời gian Delay cụ thể.

Với mục đích đó, chúng ta có thể dễ dàng biết các Property của SimpleTrigger bao gồm: Một thời điểm kích hoạt, số lần lặp lại, chu kỳ lặp lại. Tất cả những Property đó chính xác là những gì chúng ta cần, với một chút lưu ý liên quan tới thời điểm kết thúc.

Số lần lặp lại có thể là 0, hoặc một số nguyên dương nào đó, hoặc Constant Value SimpleTrigger.RepeatIndefinitely. Repeat Interval Property phải có giá trị là TimeSpan.Zero hoặc một TimeSpan Value dương. Lưu ý rằng nếu Repeat Interval có giá trị là 0 sẽ gây ra “Repeat Count” kích hoạt Trigger xảy ra một cách đồng thời (hoặc gần như đồng thời được Scheduler kiểm soát).

EndTimeUtc Property (nếu được chỉ định cụ thể) sẽ override Repeat Count Property. Việc này hữu ích nếu chúng ta mong muốn tạo một Trigger giống kiểu kích hoạt mỗi 10 giây cho tới một lúc nào đó hơn là việc chỉ định Trigger này thực hiện bao nhiêu lần. Chúng ta có thể đơn giản chỉ định EndTimeUtc với Repeat Count với giá trị RepeatIndefinitely.

Simple Trigger được build dựa trên TriggerBuilder và WithSimpleSchedule Extension Method

Ví dụ build một Trigger cho một khoảng thời gian, không lặp lại

// trigger builder creates simple trigger by default, actually an ITrigger is returned

ISimpleTrigger trigger = (ISimpleTrigger) TriggerBuilder.Create()

.WithIdentity("trigger1", "group1")

.StartAt(myStartTime) // some Date

.ForJob("job1", "group1") // identify job with name, group strings

.Build();

Build một Trigger cho một khoảng thời gian, lặp lại sau mỗi 10 giây

ITrigger trigger = TriggerBuilder.Create()

.WithIdentity("trigger3", "group1")

.StartAt(myTimeToStartFiring) // if a start time is not given (if this line were omitted), "now" is implied

.WithSimpleSchedule(x => x

.WithIntervalInSeconds(10)

.WithRepeatCount(10)) // note that 10 repeats will give a total of 11 firings

.ForJob(myJob) // identify job with handle to its JobDetail itself

.Build();

Build một Trigger kích hoạt một lần, xảy ra trong 5 phút sau

ITrigger trigger = TriggerBuilder.Create()

.WithIdentity("trigger5", "group1")

.StartAt(DateBuilder.FutureDate(5, IntervalUnit.Minute)) // use DateBuilder to create a date in the future

.ForJob(myJobKey) // identify job with its JobKey

.Build();

Build một Trigger kích hoạt ngay bây giờ, lặp lại sau mỗi 5 phút, cho tới 22:00

ITrigger trigger = TriggerBuilder.Create()

.WithIdentity("trigger7", "group1")

.WithSimpleSchedule(x => x

.WithIntervalInMinutes(5)

.RepeatForever())

.EndAt(DateBuilder.DateOf(22, 0, 0))

.Build();

Build một Trigger kích hoạt tại giờ tiếp theo, lặp lại mỗi 2 giờ một lần mãi mãi

ITrigger trigger = TriggerBuilder.Create()

.WithIdentity("trigger8") // because group is not specified, "trigger8" will be in the default group

.StartAt(DateBuilder.EvenHourDate(null)) // get the next even-hour (minutes and seconds zero ("00:00"))

.WithSimpleSchedule(x => x

.WithIntervalInHours(2)

.RepeatForever())

// note that in this example, 'forJob(..)' is not called

// - which is valid if the trigger is passed to the scheduler along with the job

.Build();

await scheduler.scheduleJob(trigger, job);

#### SimpleTrigger Misfire Instruction

SimpleTrigger có một và Instruction có thể sử dụng để thông báo cho Quartz.NET nên làm gì nếu một Misfire xảy. Những Instructions được định nghĩa là hằng số tại MisfirePolicy.SimpleTrigger.

Misfire Instruction Constant cho SimpleTrigger

* MisfireInstruction.IgnoreMisfirePolicy
* MisfirePolicy.SimpleTrigger.FireNow
* MisfirePolicy.SimpleTrigger.RescheduleNowWithExistingRepeatCount
* MisfirePolicy.SimpleTrigger.RescheduleNowWithRemainingRepeatCount
* MisfirePolicy.SimpleTrigger.RescheduleNextWithRemainingCount
* MisfirePolicy.SimpleTrigger.RescheduleNextWithExistingCount