

DotNetNuke Scheduling Provider

Dan Caron



Version 1.0.0

Last Updated: June 20, 2006

Category: Scheduler



DotNetNuke Scheduling Provider

Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, places, or events is intended or should be inferred.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Perpetual Motion Interactive Systems, Inc. Perpetual Motion Interactive Systems may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Perpetual Motion, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Copyright © 2005, Perpetual Motion Interactive Systems, Inc. All Rights Reserved.

DotNetNuke® and the DotNetNuke logo are either registered trademarks or trademarks of Perpetual Motion Interactive Systems, Inc. in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.



DotNetNuke Scheduling Provider

Abstract

In order to clarify the intellectual property license granted with contributions of software from any person or entity (the "Contributor"), Perpetual Motion Interactive Systems Inc. must have a Contributor License Agreement on file that has been signed by the Contributor.

Contents

DotNetNuke Scheduling Provider	1
Introduction	1
Ideal Solution	1
Design/Architecture Issues.....	2
Implemented Solution	2
Schedule Screen	4
Edit Schedule Screen.....	5
Schedule Status Screen	7
Schedule History Screen	9
Data Structure	9
Sample Code.....	10
Scheduled Task Settings.....	12
Class Diagram	13
Additional Information.....	14
Appendix A: Document History	15

DotNetNuke Scheduling Provider

Introduction

In DotNetNuke 2.0 we introduced two pieces of functionality that required recurring operations to be processed regularly (Users Online and Site Log). A solution was implemented that launched separate threads on Global.asax.vb in the Application_Start method for each of the operations. Functionality on the horizon for DotNetNuke will ultimately require more of these types of recurring operations. Our goal is to provide a solution that allows core functionality and 3rd party functionality to integrate easily into a DotNetNuke scheduling engine.

Ideal Solution

The ideal solution is one that allows for scheduled tasks to be run at specified intervals or scheduled times. The scheduler should run 24/7. It should allow for 3rd party modules to easily schedule tasks to be run. It should have an interface to display the current status of the scheduled tasks as well as their history. We should have the ability to edit the schedule of tasks in the scheduler. And, finally, the scheduler should be implemented using the Provider Model so competing scheduling products can be easily integrated without modifications to the DotNetNuke core.

DotNetNuke Scheduling Provider

Design/Architecture Issues

One limitation of the Scheduler is that it cannot run 24/7 without help from an external program. This is a limitation of ASP.NET, and not DotNetNuke. The Worker Process used within IIS will periodically recycle according to settings in machine.config. Some hosts may have settings that recycle the worker process every 30 minutes (forced), while some may have more complicated settings, such as recycling the worker process after 3000 web site hits, or after 20 minutes of inactivity. It is this recycling of the worker process that will shut down the scheduler, until the worker process is started again (i.e. by someone hitting the website, which in turn starts up the worker process, starting up the scheduler as well).

This functionality is actually a major benefit to web applications as a whole, in a hosted environment because it keeps runaway applications from taking down the server. But, it isn't without its drawbacks as we experience them with the scheduler.

The bottom line is that the scheduler will run 24/7, as long as someone is constantly visiting your website. It is during periods of dormancy that it possibly could shut down. It is for this reason that you need to proceed with caution in regards to the types of tasks you schedule. Make sure the tasks don't have to run "every night at midnight", etc...a more suitable task is one that runs "once per day" or "once every 2 minutes", that doesn't mind if it's not run during periods of inactivity.

Implemented Solution

The scheduling solution introduced in DotNetNuke 2.1.1 is a multi-threaded scheduler that utilizes a thread pool to manage the tasks. The thread pool helps to reuse threads that have recently been used. So, rather than killing and creating new threads the thread pool reuses them.

Creating a multi-threaded application is quite tricky, as you have to take great care in making sure that no two threads can write to the same object simultaneously. To reach a reliable multi-threaded application, there are several instances of a ReaderWriterLock that help to lock/unlock objects for read/write access.

First let's see how web.config sets up the scheduler to run.

DotNetNuke Scheduling Provider

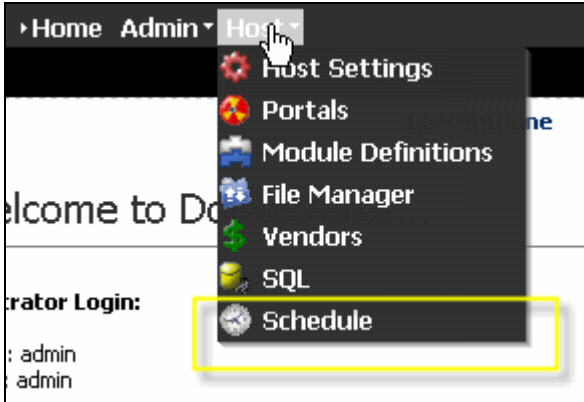
```
<scheduling defaultProvider="DNNScheduler" >
  <providers>
    <clear/>
    <add name = "DNNScheduler"
      type = "DotNetNuke.Scheduling.DNNScheduler, DotNetNuke.DNNScheduler"
      providerPath = "~\Providers\SchedulingProviders\DNNScheduler\"
      debug="false"
      maxThreads="-1"
      enabled="true"
    />
  </providers>
</scheduling>
```

You can see the scheduler uses the provider model, similar in setup to the data providers. Note the following attributes:

- ❖ `debug` – this setting, when set to true, will generate a lot of log file entries to help debug the scheduler. Debugging multi-threaded applications is always a challenge. This is one setting that can help you figure out why a task is or isn't getting run.
- ❖ `maxThreads` – this specifies the maximum number of threads to use for the scheduler. "-1" is the default in web.config, which means "leave it up to the scheduler to figure out". In the scheduler, it is capped at 10 as a max. If you specify a value greater than 0, it will use that number as the max # of thread pools to use.
- ❖ `enabled` – this is the high-level power switch for the scheduler. Set to "false" to disable scheduled events entirely. This is helpful if you are debugging other DotNetNuke functionality...it helps alleviate the multi-threaded debugging challenge when it's not necessary.

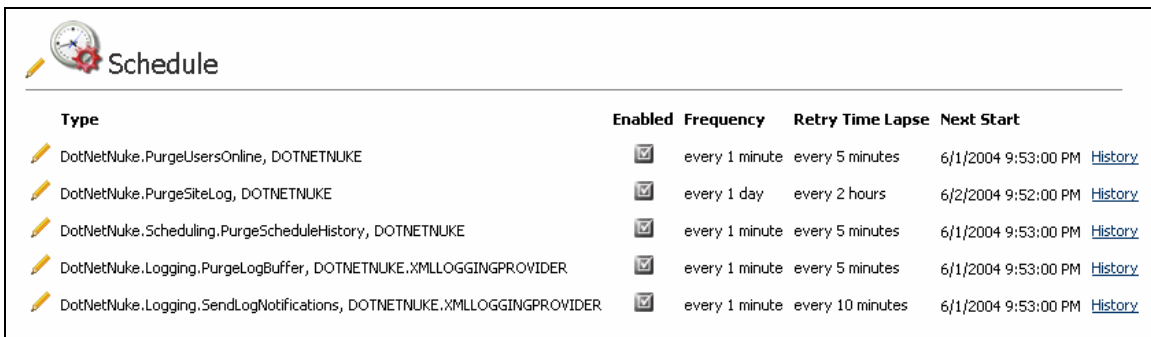
Now let's look at where to find the scheduler in DotNetNuke. Here is where it is:

DotNetNuke Scheduling Provider








Schedule Screen

Clicking on Schedule will bring you to the following screen. This screen shows you the Type (more on that later), whether the scheduled task is enabled or not, the frequency to run the task, the retry time lapse (used if the task fails), and the next start date & time for that task. There are two links for each task: a) an edit icon b) a history link. More on these later in this document.

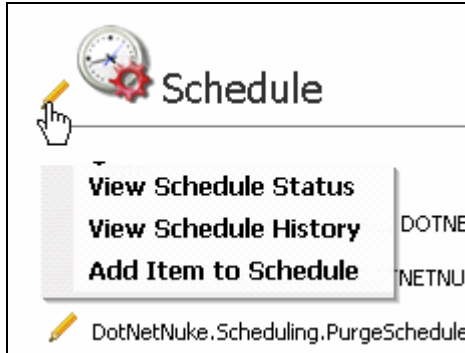


The screenshot shows the 'Schedule' screen with a table of tasks. Each row includes an edit icon (pencil), the task name, an 'Enabled' checkbox, 'Frequency', 'Retry Time Lapse', 'Next Start' date and time, and a 'History' link.

Type	Enabled	Frequency	Retry Time Lapse	Next Start	
 DotNetNuke.PurgeUsersOnline, DOTNETNUKE	<input checked="" type="checkbox"/>	every 1 minute	every 5 minutes	6/1/2004 9:53:00 PM	History
 DotNetNuke.PurgeSiteLog, DOTNETNUKE	<input checked="" type="checkbox"/>	every 1 day	every 2 hours	6/2/2004 9:52:00 PM	History
 DotNetNuke.Scheduling.PurgeScheduleHistory, DOTNETNUKE	<input checked="" type="checkbox"/>	every 1 minute	every 5 minutes	6/1/2004 9:53:00 PM	History
 DotNetNuke.Logging.PurgeLogBuffer, DOTNETNUKE.XMLLOGGINGPROVIDER	<input checked="" type="checkbox"/>	every 1 minute	every 5 minutes	6/1/2004 9:53:00 PM	History
 DotNetNuke.Logging.SendLogNotifications, DOTNETNUKE.XMLLOGGINGPROVIDER	<input checked="" type="checkbox"/>	every 1 minute	every 10 minutes	6/1/2004 9:53:00 PM	History


The following actions are available from this screen:

DotNetNuke Scheduling Provider



Edit Schedule Screen

You can edit a scheduled task's settings by clicking on the pencil next to the task. This will bring you to the following screen:

 **Edit Schedule**

Available Tasks:

Schedule Enabled: Yes

Time Lapse:
Example: "5" and select "Minutes" to run task every 5 minutes. Leave blank to disable timer for this task.

Retry Frequency:
Example: "5" and select "Minutes" to retry the task every 5 minutes after a failure. Leave blank to disable retry-timer for this task.

Retain Schedule History:
Example: Select "10" to keep the ten most recent schedule history rows.

Run on Event:
Example: Select "Application Start" to run this event when the web app starts. Please note, events run on APPLICATION_END may not run reliably on some hosts.

Catch Up Enabled: Yes
If checked, if the webserver is ever out of service, when the webserver is back in service this event will run once for each frequency that was missed during the downtime.

Object Dependencies:
Enter the tables or other objects that this event is dependent on. Example: "SiteLog,Users,UsersOnline"

[Delete](#) [Save](#)


- ❖ Available Tasks – this drop down includes a list of all classes in any assemblies in the /bin directory that inherit from DotNetNuke.Scheduling.SchedulerClient. Reflection is used to gather this list.

DotNetNuke Scheduling Provider

- ❖ Schedule Enabled – select this to enable the task to run in the scheduler. Uncheck it to disable the task in the scheduler.
- ❖ Time Lapse – this represents how often you would like the task to run. You may choose “x” number of minutes/hours/days.
- ❖ Retry Frequency – if the scheduled task fails, it will be retried after this timeframe has lapsed.
- ❖ Retain Schedule History – each time the task is run, a record is stored in the database to reflect the success/failure of the task, and it also stores any notes that were written during execution (more on that later). The number specified in this field represents how many records to retain in history for this task.
- ❖ Run on Event – You can schedule tasks to run on APPLICATION_START. Currently the only option here is APPLICATION_START. After quite a bit of testing, it was discovered that APPLICATION_END is a bad place to put code that must run...there is no guarantee that it will ever run...therefore APPLICATION_END is not an option here.
- ❖ Catch up Enabled – If your task is scheduled to run every 10 minutes, for instance, and the scheduler is shut down for some reason (reboot of server, etc...). When the scheduler is started again, if catch up is enabled for this task, the task will run once for each of the time lapses that were missed. So if the scheduler was down for an hour and catch up is enabled, it will run the task 6 times to catch up. If catch up is not enabled, the scheduler will just run the scheduled task once and continue with its schedule
- ❖ Object Dependencies – Since the scheduler is multi-threaded, it is important to avoid deadlocks on simultaneously running threads. For this reason, an object dependency can be specified to prevent other tasks with the same object dependency at the same time. For instance, if you have one task (“Task A”) that does a select on the Users table...and it has an object dependency of “Users” (this doesn’t necessarily have to relate to a table name, it can be anything, but for clarity I’m using “Users” because we are using the Users table)...another task (“Task B”) does an massive update on the “Users” table. If you don’t want these two tasks to run at the same time ever, then make sure they have the same object dependency. You can specify more than one (comma delimited). So for “Task B” you could have object dependencies of “Users,UsersOnline,Portals”...and that task won’t run when “Task A” is running because they have conflicting dependencies. One will run, and when it finishes, the other will run.

Schedule Status Screen

Clicking on “View Schedule Status” will bring you to the following screen. In this screen you can see the current status of the schedule, how many threads are active/available, and you can start and stop the scheduler. You can also see any tasks that are currently processing, as well as those in the queue.

 **Schedule Status**

Current Status:	RUNNING_TIMER_SCHEDULE
Max Threads:	10
Active Threads:	2
Free Threads:	8
Command:	Start Stop

Items Processing

Schedule ID	Type	Started	Duration (seconds)	Object Dependencies	Triggered By	Thread	Notes	Process Group
3	DotNetNuke.Scheduling.PurgeScheduleHistory, DOTNETNUKE	6/1/2004 10:28:47 PM	0.0100144	ScheduleHistory	STARTED_FROM_TIMER	841		0
5	DotNetNuke.Logging.SendLogNotifications, DOTNETNUKE.XMLLOGGINGPROVIDER	6/1/2004 10:28:47 PM	0	XMLLoggingProvider	STARTED_FROM_TIMER	994		4

Items in Queue

Schedule ID	Type	Next Start	Overdue (seconds)	Time Remaining (seconds)	Object Dependencies	Triggered By	Process Group
2	DotNetNuke.PurgeSiteLog, DOTNETNUKE	6/2/2004 9:52:00 PM		84193.025768	SiteLog	STARTED_FROM_TIMER	Unassigned
4	DotNetNuke.Logging.PurgeLogBuffer, DOTNETNUKE.XMLLOGGINGPROVIDER	6/1/2004 10:28:43 PM	4.0621168	0	XMLLoggingProvider	STARTED_FROM_TIMER	4
1	DotNetNuke.PurgeUsersOnline, DOTNETNUKE	6/1/2004 10:28:43 PM	4.042088	0	UsersOnline	STARTED_FROM_TIMER	8

❖ **Current Status** – this tells you what status the scheduler is in. Values you may see here are:

WAITING_FOR_OPEN_THREAD

DotNetNuke Scheduling Provider

RUNNING_EVENT_SCHEDULE
RUNNING_TIMER_SCHEDULE
SHUTTING_DOWN
STOPPED

❖ Max Threads – this is determined in web.config (details above)
Active Threads – tells you how many tasks are currently running. Above you can see two tasks running.

- ❖ Free Threads – this is MaxThreads minus Active Threads
- ❖ Command – this allows you to start/stop the scheduler. Note: you can disable it in web.config as well for a long-term setting.
- ❖ Items processing – these tasks are currently being executed.
 - Schedule ID – unique identifier for the scheduled task
 - Type – the fully qualified type & assembly of the task
 - Started – time & date when the task was started
 - Duration - # of seconds the task has been running
 - Object dependencies – explained above
 - Triggered by – tells you whether the task was triggered by an event or the timer
 - Thread – this is the thread id that the task is running on
 - Notes – any notes that are written out during task execution will be displayed here (more on this later)
 - Process group – this is the numeric representation of the thread pool that the task was assigned to (helpful for multi-threaded debugging, etc.)
- ❖ Items in Queue – these tasks are queued up for execution.
 - Schedule ID – unique identifier for the scheduled task
 - Type – the fully qualified type & assembly of the task
 - Next Start – time & date when the task is scheduled to run next
 - Overdue - # of seconds that have passed since the task should have run
 - Time Remaining - # of seconds until the task is scheduled to run
 - Object dependencies – explained above
 - Triggered by – tells you whether the task was triggered by an event or the timer
 - Process group – this is the numeric representation of the thread pool that the task was assigned to (helpful for multi-threaded debugging, etc.). The thread pool is assigned just before the task is executed.

DotNetNuke Scheduling Provider

Schedule History Screen

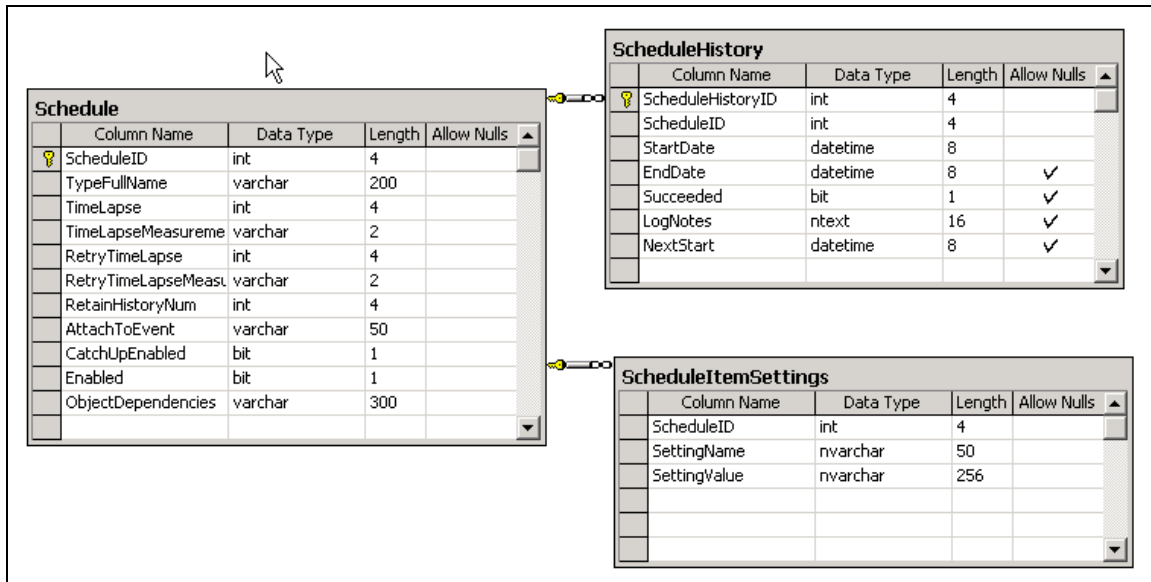
The following screen represents the history of a scheduled task. You can see when the task began, ended, how long it took to execute, whether it was successful, when its next start was scheduled for, and any notes written out during execution.

Schedule History					
Started	Ended	Duration (ms)	Succeeded	Next Start	Notes
6/1/2004 9:52:00 PM	6/1/2004 9:52:04 PM	3.976	True	6/2/2004 9:52:00 PM	Purged Site Log Successfully
5/31/2004 12:27:02 PM	5/31/2004 12:27:03 PM	1.382	True	6/1/2004 12:27:02 PM	Purged Site Log Successfully
5/30/2004 1:25:14 AM	5/30/2004 1:25:15 AM	0.981	True	5/31/2004 1:25:14 AM	Purged Site Log Successfully

Data Structure

The data that drives the scheduler is stored in a database and utilizes the default DotNetNuke database provider as specified in web.config. There are three tables:

DotNetNuke Scheduling Provider



Sample Code

Below is an example of a scheduled tasks. It can be found in the following file in the DotNetNuke project:

`/admin/Users/UsersOnlineDB.vb`

Each task has it's own class, in this case the class name is `PurgeUsersOnline`.

The class must inherit from `DotNetNuke.Scheduling.SchedulerClient`

The class must have a constructor with the same signature as the one to the right. You must also set the value of the `ScheduleHistoryItem` to the value of the incoming parameter.

The `DoWork` method is what gets called from the scheduler. It must be follow the same logic & format as the example to the right...meaning, you need a Try/Catch, with all of the required items included (required items are noted in the code).

`Me.Progressing` tells the scheduler that the task is progressing. This is optional, and is useful for long

DotNetNuke Scheduling Provider

running tasks.

UpdateUsersOnline() is where the actual work is getting done.

Me.ScheduleHistoryItem.Succeeded = true is required after the work is done

If you want to add any notes to the schedule history, you can call “AddLogNote()” as shown to the right. You can call it as many times as necessary.

It is important to properly handle exceptions in DoWork(). Ideally, copy the entire Catch section to your task class.

Once you compile your class and put it in /bin, it will be available in the schedule admin screens to add the task to the schedule

```
353 | Class PurgeUsersOnline
354 |     Inherits DotNetNuke.Scheduling.SchedulerClient
355 |
356 |     Public Sub New(ByVal objScheduleHistoryItem As DotNetNuke.Scheduling.ScheduleHistoryItem)
357 |         MyBase.new()
358 |         Me.ScheduleHistoryItem = objScheduleHistoryItem
359 |     End Sub
360 |     Public Overrides Sub DoWork()
361 |         Try
362 |
363 |             'notification that the event is progressing
364 |             Me.Progressing() 'OPTIONAL
365 |
366 |             UpdateUsersOnline()
367 |
368 |             Me.ScheduleHistoryItem.Succeeded = True 'REQUIRED
369 |
370 |             Me.ScheduleHistoryItem.AddLogNote("Purged Users Online Successfully")
371 |
372 |         Catch exc As Exception 'REQUIRED
373 |
374 |             Me.ScheduleHistoryItem.Succeeded = False 'REQUIRED
375 |
376 |             Me.ScheduleHistoryItem.AddLogNote("EXCEPTION: " + exc.ToString) 'OPTIONAL
377 |
378 |             'notification that we have errored
379 |             Me.Errorred(exc) 'REQUIRED
380 |
381 |             'log the exception
382 |             LogException(exc) 'OPTIONAL
383 |         End Try
384 |     End Sub
385 |     Private Sub UpdateUsersOnline()
386 |
387 |         Dim objUserOnlineController As UserOnlineController = New UserOnlineController
388 |
389 |         ' Is Users Online Enabled?
390 |         '
391 |         If (objUserOnlineController.IsEnabled()) Then
392 |             ' Update the Users Online records from Cache
393 |             '
394 |             Me.Status = "Updating Users Online"
395 |             objUserOnlineController.UpdateUsersOnline()
396 |             Me.Status = "Update Users Online Successfully"
397 |             Me.ScheduleHistoryItem.AddLogNote("Users Online Updated Successfully")
398 |             Me.ScheduleHistoryItem.Succeeded = True
399 |         End If
400 |
401 |     End Sub
402 | End Class
```

Scheduled Task Settings

There is a table named “ScheduleItemSettings” that can store settings for each scheduled task. The settings are stored in key/value pairs with a foreign key of Schedule.ScheduleID. The settings can be retrieved from your task class (i.e. in the DoWork() method perhaps) using the following syntax:

```
Dim myValue as String = Me.ScheduleHistoryItem.GetSetting("MyKey")
```

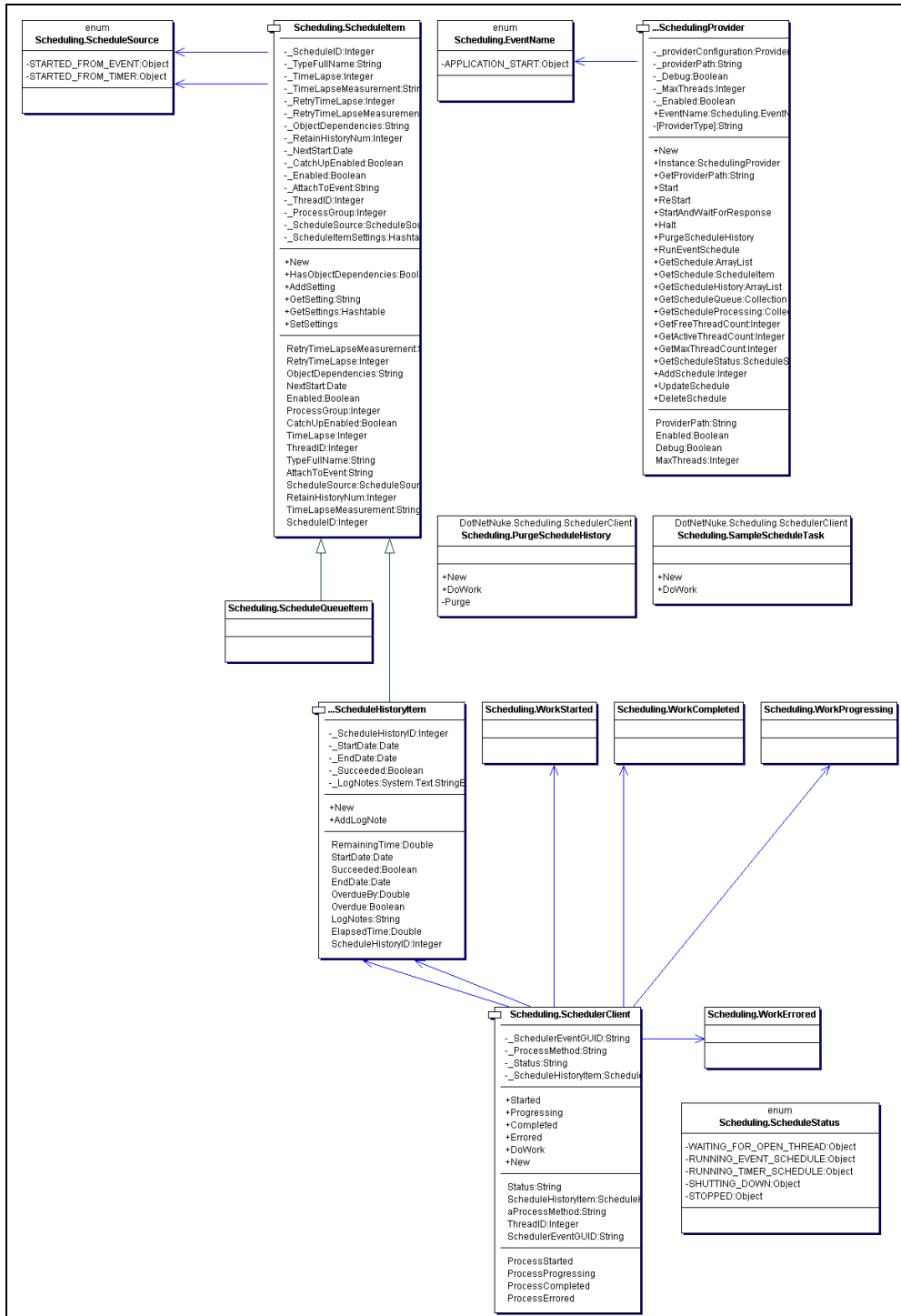
Or you can retrieve the entire collection of settings in a HashTable:

```
Dim myHashTable as HashTable = Me.ScheduleHistoryItem.GetSettings
```

The settings are retrieved from the table when the schedule queue is refreshed from the database (every 10 minutes or when a change is made to the schedule).

DotNetNuke Scheduling Provider

Class Diagram



Additional Information

The DotNetNuke Portal Application Framework is constantly being revised and improved. To ensure that you have the most recent version of the software and this document, please visit the DotNetNuke website at:

<http://www.dotnetnuke.com>

The following additional websites provide helpful information about technologies and concepts related to DotNetNuke:

DotNetNuke Community Forums

<http://www.dotnetnuke.com/tabid/795/Default.aspx>

Microsoft® ASP.Net

<http://www.asp.net>

Open Source

<http://www.opensource.org/>

W3C Cascading Style Sheets, level 1

<http://www.w3.org/TR/CSS1>

Errors and Omissions

If you discover any errors or omissions in this document, please email marketing@dotnetnuke.com. Please provide the title of the document, the page number of the error and the corrected content along with any additional information that will help us in correcting the error.

Appendix A: Document History

Version	Last Update	Author(s)	Changes
1.0.0	Aug 16, 2005	Shaun Walker	<ul style="list-style-type: none">Applied new template