NQueue

Batch Processing Framework – Users Guide

Version 1.0

*Contents Page*

1.1 Goals of NQueue 3

2 NQueue Architecture 4

2.1 Overview 4

2.2 Getting Started 5

2.3 Installation 7

2.4 Software Architecture 9

3 Technical Guide 11

3.1 Adding new Jobs 11

3.2 Adding new hosts and host instances 12

3.3 Schedule Processing 13

3.4 Job Dequeuing 15

3.5 Job Execution 16

3.6 Database Design 17

4 Appendix A: Design Patterns 18

4.1 Run work in a resilient manner 18

4.2 Multiple jobs to do similar work and/or in multiple environments 18

4.3 Perform work with particular credentials 18

4.4 Redirecting to a new version of a job definition dll 18

4.5 Batch job shells out to an external command line executable 18

4.6 External system pushes file to local system for import 19

5 Appendix B: Product Backlog 20

6 Appendix C: Framework Code; Visual Studio Solution 21

*Figures*

[Figure 1: Architecture Overview 4](#_Toc261881743)

[Figure 2: Software Architecture 10](#_Toc261881744)

[Figure 6 Control Single Host Instance dialog 13](#_Toc261881745)

[Figure 8 Association between host instance and work items 16](#_Toc261881746)

Introduction

NQueue is intended to provide a work scheduling and execution framework and toolset that contains no single point of failure. Using a farm of servers (the 2 or more servers you've already put in place to provide redundant processing for a website, BizTalk, SQL, etc) and a clustered SQL server backend, multiple NQueue windows services compete to evaluate configured schedules and execute work.

NQueue is reasonably stable for execution in a production environment at the moment but is lacking some obvious items of functionality. See Appendix B: Product Backlog for more information.

## Goals of NQueue

The following points are design goals of NQueue:

* Schedule processing of tasks (scheduling is handled by the framework not by an external scheduler)
* Introduce no single point of failure (scheduling or tasks)
* Run tasks with arbitrary credentials
* Call arbitrary .NET code
* Full and persistent logging of batch execution
* Resilient (no SPOF and framework code runs in isolation from batch code)
* Operable (tools)
* Tasks can be scheduled from the command line (i.e. invoked externally)
* No reliance on other tools, frameworks (NHibernate, EntLib, etc)

The following points are *not* currently design goals of NQueue

* Exact scheduling. The distributed execution and polling means that jobs can't be guaranteed to execute to the minute.
* Failed execution and checkpointing. Unlike BizTalk (from which the general architecture has been copied) jobs are treated as atomic actions which may or may not execute without exception, but so long as the job is dequeued and called successfully, NQueue has successfully run the job.

# NQueue Architecture

## Overview

NQueue is a distributed system written in C# composed of the following high level components;

* Admin tool – A windows form application that enables add and delete operations against the various artefacts in a NQueue installation (jobs, schedules, host instances). This would typically be used to view or change the system configuration although it can also be used to immediately enqueue items of work for execution (during testing for example)
* NQueue Monitor website – A website allowing operations and support staff to view the progress of configured jobs. They may also pause or disable job instances from this tool.
* SQL Database – all state for the system is stored in a central (clustered) SQL database.
* Windows services – NQueue processing services running on any number of configured servers competing to evaluate job schedules and execute job code.
* Client API – A .net class library that users can consume/inherit from to allow their job code to interact with the framework.
* NQueueCmd – command line enqueuing of work to execute immediately.



Figure : Architecture Overview

## Getting Started

If you are a developer wanting to get NQueue up and running and discover how to develop against it, follow these steps to install and configure an NQueue host instance, create a job and run it.

Note that this system works well on server platforms - Windows 2003, 2008 - but security issues around installing and running new services, etc in Vista and Windows 7 mean that some things don't work so well on those platforms. You can run the test service (and command line application that runs the same code as the windows service) to test things in Vista and Windows 7.

1. As per 2.3.2, create the SQL database by running the 4 scripts.
2. Add your windows login to the NQueue\_administrators database role either directly or via group membership
3. Create a windows login to run services under and run the web admin site. Add this login to the NQueue\_system database role either directly or via group membership
4. As per 2.3.2, install the main installer. Download or compile NQueue.Setup and install it to your workstation. Remember to edit the .config files to point to the correct database if this is not (local). Try running the admin console. Close it.
5. Create an app pool "NQueueAppPool" in IIS with the NQueue services identity. Don't forget to add that identity to the local IIS\_WPG group.
6. Install the web admin site installer. Download or compile NQueue\_BatchAdmin.setup and install to your workstation. Pick the NQueueAppPool app pool when prompted.
7. Edit the web.config file at the installed location to change (local) in connection strings for your SQL server location if necessary. Try the site out - it shouldn't be very interesting yet.
8. Create a job definition - create a class library project, referencing NQueue.Client.dll (from where you installed it) with the following code. Read section 3.1 for more information on authoring jobs but for now the following code demonstrates the framework.

using System;

using NQueue.Client;

namespace TestJob

{

public class HelloWorldJob : Job

{

protected override ExecutionResults

ExecuteJobImplementation(ExecutionContext context)

{

if (!(ParametersObject is TestJobParams))

{

throw new InvalidOperationException(

"Parameters object was not of type TestJobParams");

}

TestJobParams jobParams = (TestJobParams)ParametersObject;

System.Diagnostics.EventLog.WriteEntry("NQueue",

"Hello " + jobParams.Name);

return ExecutionResults.Success;

}

protected override string JobTypeName

{

get { return "Test Job"; }

}

public override Type ParametersType

{

get { return typeof(TestJobParams); }

}

}

public class TestJobParams

{

private string \_name;

public string Name

{

get { return \_name; }

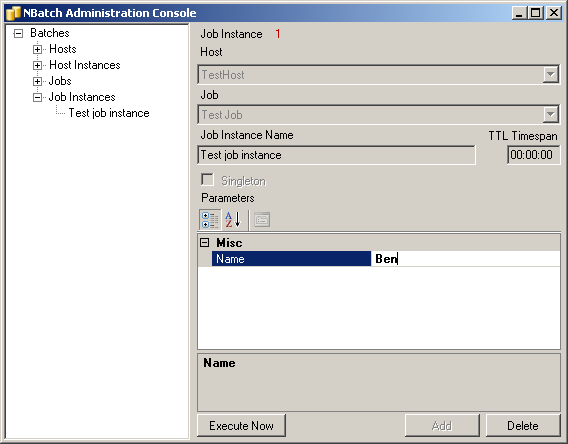
set { \_name = value; }

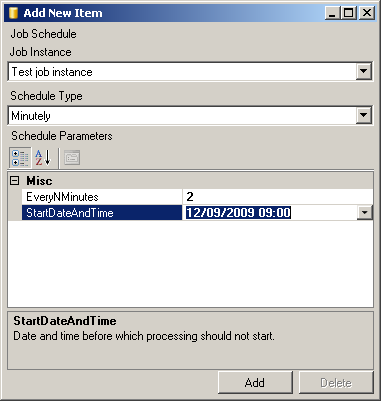
}

}

}

1. Compile and GAC this dll. Your batch code must be GACed or installed under the NQueue install directory for the framework to find it.
2. In the NQueue administration console, right click in the tree on the left and select Add New -> Job Definition. Give a name and description
3. Next to Job Assembly, click the ellipsis to launch the type browser. Click the folder icon to select your assembly (from the file system), click the tick icon and double click the HelloWorldJob class.
4. Click Add.
5. Click Add New -> Host. Call the host "TestHost".
6. Click Add New -> Host Instance. Select TestHost and type the local machine name.
7. Select the host instance in the tree on the left and click Control Instance. In the Control Host Instance Service dialog, click Refresh, then click Install.
8. Give the credentials of the service identity created in step 3, and click ok.
9. Give the credentials logon as a service rights
10. Click start
11. Click Add New -> Job Instance
12. Select the test host and job. Give an instance name of "Job instance test"
13. Type your name in the property grid and click Add.



1. Click Execute Now. The job should execute and write your name to the event log. You should be able to browse the log of the job executing in the admin site.
2. Right click and select Add New -> Schedule
3. Create a minutely schedule for the test job, and set it to execute every 2 minutes. Set the start date and time to be in the past (TODO: Need a datetime picker here - you can pick the date but have to type the time). Click Add.  
   
4. The job should now run by itself every 2 minutes (although it'll be a bit out at the start but should settle down to every two minutes). Verify this in the admin site and in the event log. If the job does not execute you can debug the service by running NQueue.Service.Test.exe which can impersonate the windows service and provides detailed tracing information.

## Installation

Full installation details for development machines or production servers.

### Installing the NQueue code base to edit

If you wish to install the NQueue code base to review, edit or debug, perform the following steps to install and verify NQueue. If you are installing to a runtime environment see section 2.3.2.

* Extract the solution folder to your local drive (NQueue)
* Open the solution and compile with visual studio
* On the SQL server you will be using run the database setup scripts to create the database (NQueue), tables, stored procedures and seed data the base install requires. The following scripts should be run;
  + 0. CreateNQueueDatabase.sql
  + 1. CreateNQueueTables.sql
  + 2. CreateNQueueStoredProcedures.sql
  + 3. CreateNQueueSeedData.sql
* In SQL Server, add your windows account to the NQueue\_administrator role so that you have permissions to run the admin tool, etc.
* If the SQL server you will be using is not local, do a replace in files in visual studio for (local) and replace with your SQL Server name (and instance) - this swaps the values out in the connection strings in the various config files.
* Run the windows form administration tool (project name is simply "NQueue"). This verifies connection to the database and also adds the NQueue event source to the event log.
* Run the unit tests project and verify the unit tests pass (NQueue.nunit in NQueue.UnitTests) - you require NUnit 2.5.2 or later.
* Run the final database script provided (CleanupDatabase.sql) so that the unit test data is erased
* Note that to develop code that runs within the framework, you should still execute the relevant parts of 2.2.2

### Installing the framework in production or to develop against

For use if you are installing the framework on a development machine to write and test jobs against, *or for runtime environments where you will be executing job code*.

You will need to create windows groups and/or users to run at least one NQueue service under and to run as the identity for the web admin site application pool. These should be added to the NQueue\_system database role (discussed later).

Install the system in the following way

* On each processing server or development workstation, run the main installer (NQueue.Setup). This installs the winforms admin tool, the binaries for the host instance services (without actually configuring any services to run) and the client API. At the install location, edit the three .config files to point to the database you will be using [TODO: this should be set by the installer].
* On the SQL server (or on a standalone environment), run the database setup scripts to create the database (NQueue), tables, stored procedures and seed data the base install requires. The following scripts[[1]](#footnote-1) should be run;
  + 0. CreateNQueueDatabase.sql
  + 1. CreateNQueueTables.sql
  + 2. CreateNQueueStoredProcedures.sql
  + 3. CreateNQueueSeedData.sql
* In SQL Server, map the database roles provided to the appropriate windows groups so that you and the service accounts have access;
  + NQueue\_administrators - users have full access to create and delete jobs, etc
  + NQueue\_operators - users can access the monitoring website
  + NQueue\_system - host instance accounts should be in this group

* On the webserver (or webservers for a web farm) run the installer the batch monitor tool (NQueue\_BatchAdmin.Setup). At the install location, edit the web.config file to point to the database you will be using. The website should be configured to use an app pool with an identity of a user assigned to the NQueue\_system database role. That identity needs to be in the local IIS\_WPG group.
* Any service identities (host instance accounts) need to have the logon as a service right
* A developer of batch code would reference the client API from the installed location (NQueue.Client.dll). No other assembly references are required.

For a typical collapsed build (developer scenario) the SQL database would be installed locally and the main setup executed to provide access to the application binaries. The batch monitor website would also optionally be installed.

For a production build, the SQL database would be installed on a fault tolerant SQL cluster and the batch monitor tool installed on 2 or more load balanced web servers. The main install would then be run on all servers that will be running batch jobs. The admin tool is used to install one or more instances of the NQueue service on remove computers, and then writes configuration data to the database, which determines which services on which servers perform which jobs.

## Software Architecture

The following software artifacts combine to schedule and execute work

* Schedule Types – types of schedules that can be configured within the batch framework, e.g. Daily, Weekly, Minutely, Yearly.
* Host – a logical grouping of related work is assigned to a host.
* Host instance – instance of a host on a server, expressed as a windows service installed and running on that server.
* Job – configured work that can be performed, expressed as a class within a .net assembly.
* Job Instance – an instance of a job type. An instance of a job is assigned a host on which it will execute and a set of values for the parameters that job takes. For example, a SQL maintenance job type might take parameters describing the connection string to access a SQL server and a stored procedure name to execute. The code behind this job can then be configured to execute stored procedures on any number of SQL databases through configuration of the job instance.
* Finally a job instance may be given one or more job instance schedules – an instance of a Schedule Type. For example the SQL maintenance job may be scheduled with the daily schedule type, where (for example) *the daily schedule processor is configured* to run on every week day only, or every 3rd day of the week.



Figure : Software Architecture

# Technical Guide

## Adding new Jobs

### Introduction

To author new job code for execution in the framework, create a .net assembly with a class that inherits from the provided base class. To add parameters to your job, create a class that encapsulates these parameters. The class can then be configured into the framework and installed on all servers that will execute it.

### Authoring Job code

Add a reference to NQueue.Client.dll.

Create a parameters class – this should be a public class marked with the [Serializable] attribute that encapsulates all the customisation you wish to perform on an instance of your job type (see section 3.1.3 below).

Create a new class that inherits from NQueue.Client.Job

Implement the following methods and properties;

|  |  |
| --- | --- |
| **Method\Property** | **Description** |
| ExecutionResults ExecuteJobImplementation( ExecutionContext context) | Execute the main work of the job in this method, and return an indication of the outcome of the job via the return parameter.  The ExecutionContext object passed into this method allows access to the logger object you should use to log events to the database and the Mutex Manager API |
| string JobTypeName | Property; return a short name for the job here – used in the UI |
| Type ParametersType | Property; return the type of your parameters class here. |

### Parameters class

The parameters class should contain any parameterisation you wish to use on your job class. This could be because the parameters change between environments and\or because you are creating a generic job class which can be applied to multiple job instances. Each data item you wish to persist with the class should be exposed as a public property.

The class should be marked as [Serializable] and will be XML serialised to the database when the job instance is created (using the public Getters and Setters on your properties). In the UI, the class is displayed using the .net PropertyGrid control, so you can provide descriptions for each property using the [Description] attribute.

E.g.

[Description("Folder to move the input file to if the import fails")]

public string FailedFolderPath

{

get { return \_failedFolderPath; }

set { \_failedFolderPath = value; }

}

### Installation of Job code

The assembly must be first configured into the framework. Right click on the tree on the left hand side of the administration console and select *Add New… Job Definition*. Use the UI to select the compiled assembly and use the supplied class browser to select the Job class.

Use the UI to create and configure the Job Instance by selecting *Add New… Job Instance*.

At runtime the assembly must be made available to the processing services. Currently this means either installing at the services local path or in the GAC. Ideally the assembly will be signed and installed in the GAC. This means that the reference to the assembly will contain the version and that multiple versions can be referenced, allowing for side by side deployment of batch code. On the other hand, if you wish to update the version of the code but not edit the reference in the batch framework, policy files can be installed in the GAC to redirect to the later version.

### Logging

As the work executed in the Job definitions is almost by definition performed unattended, full logging of the job’s activities and any errors during execution is encouraged. To enable this, a logger object is passed into the ExecuteJobImplementation method as part of the ExecutionContext.

E.g.

IExecutionLoggingStrategy logger = context.Logger;

logger.WriteInfo("Importing file");

The implementation passed in logs information to the database which can then be viewed in the batch administration website.

### Singletons and MutexManager

In a distributed batch execution context, it is desirable to set blocks of code or entire jobs to execute only one at a time. The MutexManager object (again, part of the execution context object passed into the ExecuteJobImplementation method) allows you to take out a lock in the database so that any other code attempting to execute at the same time is unable to take out a lock with the same name until you release it. The MutexManager implements IDisposable so you can use it in a using() statement to guarantee release of the mutex.

If a whole job should never be executed more than once in parallel (most data import jobs are like this), you can mark the entire job as a singleton (this is done at the job instance definition). This causes the framework to wrap the entire job execution in a mutex lock. If the mutex cannot be claimed the job is marked as a duplicate and your implementation code isn’t called.

Note that this simply stops *simultaneous* execution of a job instance (by multiple host instances). You should still write your jobs so that they can be called multiple times serially without issue.

## Adding new hosts and host instances

Hosts are used to group job instances together logically and provide a common set of credentials for their execution.

Host instances are instances of a host on a server, expressed as a windows service.

### Creating a Host

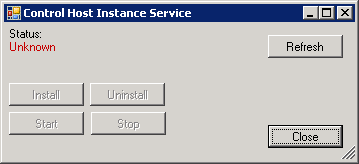
Open the batch administration console and right click in the left hand pane, select *Add New… Host*.

### Creating a Host Instance

Open the batch administration console and right click in the left hand pane, select *Add New… Host Instance.*

Provide the name of the server on which you want to install the host instance, and select Add.

Then select the new host instance in the tree view on the left, and click Control Instance;



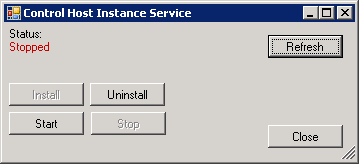
****

Figure Control Single Host Instance dialog

Then click refresh

If the service is not installed, the button “Install” will activate. Click this and provide credentials for the service. The administration console will install the service on the target server. The service can then be started from this dialog.

TODO: this UI is a bit kooky and needs making a bit more intuitive.

### Restart all host instances

All host instances configured within the group can be conveniently stopped, started or restarted using the administration console.

In the left hand treeview, right click and select “Control All Host Instances”. Click one of the buttons to stop, start or restart all host instances.

This is convenient at deployment time – you could refresh all references to GACced components through a *restart all*, or stop all activity on the NQueue database (for a backup or restore), or postpone all job execution with a *stop all*.

## Schedule Processing

### Introduction

The following table details all the schedule types that come with the base framework.

|  |  |  |
| --- | --- | --- |
| **Schedule type** | **Variants** | **Evaluated** |
| Daily | Every day  Every week day  Every *n* days | Every day |
| Minutely | Every *n*th minute | Every minute |
| Monthly | Day of month  Every *n* month  First day of month  Last day of month | Every day |
| Weekly | Day of week  Every *n* weeks | Every day |
| Yearly | Day of year  Every *n* years  First day of year  Last day of year | Every day |

Schedule type gives the name of the schedule type and how it’s selectable within the batch admin tool.

Each type is then configured for each job instance its applied to. Various parameters can be applied, specific to the schedule type. A variant is also given, making the type applied to the instance behave in a different way.

The “evaluated” column in the table below shows how often the schedule *type* is evaluated. When the type is evaluated, all job instances configured for this type have their schedule evaluated[[2]](#footnote-2), and zero to *n* work items for that job instance may be **enqueued** in the database. Each is marked with the date and time they should be dequeued.

### Resilient scheduling

All NQueue processing services on all servers compete to evaluate all schedule types. Each service periodically[[3]](#footnote-3) polls the database checking if any schedule types are due to be evaluated. If one is due, the schedule type is locked to the calling service, evaluated, and then unlocked (table *ScheduleTypes*, fields *CurrentlyProcessing*, *LastProcessed*, *ProcessingHostInstanceID*).

In this way, any server can perform scheduling and no single point of failure is introduced.

### Adding more schedule types

Custom schedule types can easily be added to the framework. To create a new schedule type, create a class that inherits from

public abstract class NQueue.BusinessServices.Scheduling.ScheduleProcessors.ScheduleProcessor

Implement the following methods\properties;

|  |  |
| --- | --- |
| **Method\Property** | **Description** |
| Type ParametersType | Readonly property which returns a type object specifying the type which the schedule processor accepts as parameters. When a job instance is scheduled to this schedule type, an instance of this type will be created, assigned its parameters and XML serialized into the database. When the job instance schedule is then run, the XML is deserialised to an instance of this type and passed to the schedule processor. |
| ScheduleProcessingDueResults IsScheduleTypeDue(DateTime? lastRunDateTime, DateTime? anticipatedNextRunDateTime) | Evaluates whether the schedule type is due to be evaluated, not due or overdue. Overdue schedules log a warning to the event log |
| DateTime GetIndicativeNextRunDateTime() | Used to tell the system when the schedule type should next run |
| List<WorkItemToEnqueue> ExecuteConcreteInstanceScheduleProcessing( JobInstanceSchedule schedule, object scheduleParametersObject) | Called by the framework for each job instance schedule. The parameters for the job instance schedule are passed in as a deserialised object which can be cast to ParametersType.  Return a list of work items to enqueue from this method. |

The schedule type must also be added to the database – a row is added that points to the assembly and class being used. The following SQL script demonstrates how to add the Daily schedule processing type;

insert ScheduleTypes (ScheduleTypeID, ScheduleType, ScheduleTypeStrongName, ScheduleTypeClassType) values (1, 'Daily', 'NQueue.BusinessServices.Scheduling, Version=1.0.0.0, Culture=neutral, PublicKeyToken=d5f1da7c501d0b59', 'NQueue.BusinessServices.Scheduling.ScheduleProcessors.DailyScheduleProcessor')

The assembly can then be made available to the processing services – either by adding to the GAC or to installation folder for NQueue.

## Job Dequeuing

A work item can be enqueued by the schedule processors as above for a certain date and time, or can be enqueued manually by the command line utility (NQueuecmd.exe) or the batch administration tool (only enqueues for “now”, i.e. execute immediately).

In all cases the work item is inserted into the WorkQueue table, and given a date and time to be executed. The work item points to a job instance, and the job instance is assigned a host. Only host instances of that host are allowed to dequeue and execute the work item.



Figure Association between host instance and work items

All processing services in the system are configured host instances (if they are not correctly configured and authorised in the batch database, they are not allowed to start). Every minute, each polls the database looking for work items in the queue that are available for dequeueing (date and time to dequeue is in the past), are assigned to the host they're an instance of, and that are not currently locked to another host instance. Using the same competing consumers pattern as the schedule processing, if any one of the servers fails, another will automatically pick up the work load. So provided there is more than one host instance configured for each host, no single point of failure is created for any job.

## Job Execution

Work that is dequeued is executed in a new AppDomain to isolate it from the processing service. Settings are copied from the processing service app domain and if a configuration file was provided in the configuration of the job, this file is provided as the “.config” file for the new appdomain.

The class is dynamically instantiated in the AppDomain and a proxy to it returned to the framework (which is why the base Job class inherits from MarshallByRefObject). The main implementation is executed remotely via the proxy and the results (the outcome) marshaled back into the calling appdomain.

Finally, the result of the work item execution is logged to the database and the appdomain is torn down.

## Database Design

### Database diagram



# Appendix A: Design Patterns

## Run work in a resilient manner

Make sure each host has an instance on more than one server.

## Multiple jobs to do similar work and/or in multiple environments

Create a parameters class that allows you to customise the work in the way you want or to the environment in which it’s executing.

Use SQL configuration to configure the job instance appropriately.

## Perform work with particular credentials

Create a new host and install a corresponding host instance on each server and run the host instances under those credentials.

## Redirecting to a new version of a job definition dll

Deploy a policy file to the GAC which redirects execution to the new version, when a client asks for a reference to the old version;

Create an XML file, PublisherPolicy.xml – e.g. this would redirect requests for Customer.Batch.Jobs for versions 3.0.0.0 to 3.0.0.2 to the version 3.1.0.0 in the GAC.

<configuration>

<runtime>

<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

<dependentAssembly>

<assemblyIdentity name="Customer.Batch.Jobs"

publicKeyToken="9def6b52b5d1bc1c"

culture="neutral" />

<bindingRedirect oldVersion="3.0.0.0-3.0.0.2"

newVersion="3.1.0.0"/>

</dependentAssembly>

</assemblyBinding>

</runtime>

</configuration>

Use the assembly linker to compile this to a dll that can be installed in the GAC

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\al.exe /link:..\bin\Release\PublisherPolicy.xml /out:..\bin\Release\**policy**.**3.0.Customer.Batch.Jobs.dll** /keyfile:..\..\key.snk /version:3.0.0.0

The name of the output file is significant – it must follow the above format for .net binding to pick it up as the policy file for the requested assembly.

Deploy the output dll to the GAC. When you deploy the dll, the original xml file (i.e. PublisherPolicy.xml) must be in the same folder as the dll (i.e. policy.3.0.Customer.Batch.Jobs.dll) at the time you deploy. If it is not in the same folder, the deploy will fail.

## Batch job shells out to an external command line executable

Use the following code within your batch job code to shell out to an external command line executable. The output from the application is returned as a string, which can then be written to the framework information log using the logger object.

/// <summary>

/// Executes an external batch file and returns the output

/// </summary>

/// <param name="executableFilePath"></param>

private string ExecuteExternalBatchFile(string executableFilePath, int timeout)

{

FileInfo executableFileInfo = new FileInfo(executableFilePath);

System.Diagnostics.ProcessStartInfo psi = new System.Diagnostics.ProcessStartInfo(

executableFilePath);

psi.WorkingDirectory = executableFileInfo.DirectoryName;

psi.RedirectStandardOutput = true;

psi.CreateNoWindow = true;

psi.UseShellExecute = false;

System.Diagnostics.Process process = System.Diagnostics.Process.Start(psi);

System.IO.StreamReader processOutput = process.StandardOutput;

process.WaitForExit(timeout);

if (process.HasExited)

{

return processOutput.ReadToEnd();

}

else

throw new TimeoutException("Process call timed out");

}

## External system pushes file to local system for import

A common requirement is to have a file pushed from an external system to be imported by your system.

To perform this, the external system can send to more than one server, then call NQueuecmd to enqueue the job with local affinity on all servers. Set the job to run as a singleton so that only one of the invoked jobs is allowed to execute. The import itself should be idempotent in case the job runs very quickly (one invocation finishes before the next starts so that the singleton checking is not effective). If the job is sufficiently long running it should run only once, however.

In this way, if one of the servers is down the import can still take place on a surviving server without the calling system being modified.

# Appendix B: Product Backlog

The following items either should, or could be implemented in the future.

* Proper installers; accepts connection strings, etc and installs SQL database.
* Admin website needs prettying up and the CSS sorted out.
* Edit\update\save functionality in administration console; only Add and Delete currently provided.
* Add New popup menu is not context sensitive (e.g. when you right click a job instance and select add new -> schedule it should default the schedule to hang off the selected job instance)
* Cleanup SQL agent jobs are required to remove log information and old job audits - at the moment the NQueue database will grow without being cleaned up.
* Boundary conditions for schedule execution. For example, the daily job scheduler is only evaluated once a day, so if you add a new schedule to process today and the daily schedule has already been evaluated then today's job won't be queued. This isn't what people expect.
* State provider; provide API on top of the batch database (hooked onto execution context) to store state on behalf of jobs between executions.
* Tool to generate installation SQL script from database (i.e. pick artifacts to export – jobs, hosts, etc and generates SQL script. Probably wouldn’t do this host instances as they are environment specific and involve installing windows services).
* Work items exceeding the time to live or started but not (marked as) completed due to *server failure* are not dealt with currently (although failed job code is dealt with correctly). Work items exceeding the TTL should be marked as timed out and perhaps configuration could determine if the item should be requeued.
* Alerts on failure (currently provided for by standard alerting on the event log)
* Some standard jobs could be defined in a general way, well parameterised and included in the box; e.g. execute stored procedure, executed SSIS package, do HTTP post or get, send email, copy file.
* Chaining jobs?

# Appendix C: Framework Code; Visual Studio Solution

The following projects make up the visual studio solution

### Top Level Items

Database scripts – install and configures a vanilla NQueue database

Reference to the NUnit dll for unit testing.

### NQueue\_BatchAdmin

Batch Monitor website project.

### NQueue

Winform batch administration tool project.

### NQueue.BusinessServices.Jobs

Business tier logic for Job and Host processing; dequeues and invokes jobs

### NQueue.BusinessServices.Scheduling

Business tier logic for Schedule types and management and processing of schedules

### NQueue.BusinessServices.System

Various system helper functions – e.g. installs batch processing services on remote computers, starts\stops services.

### NQueue.Client

Client API a batch job developer would reference to author job definitions for the framework. In particular, all custom batch jobs should inherit from

public abstract class NQueue.Client.Job : MarshalByRefObject

See section 3.1 for more information.

### NQueue.DataServices

Database access layer for the system – all ADO.net code is encapsulated in this assembly. Also contains helper functions for XML serialising and deserialising objects.

### NQueue.Entities

Classes representing the software entites for the system.

### NQueue.Service

This project contains the windows service code that expresses the batch processing services

### NQueue.Service.Server

A dll project that contains the implementation of the batch processing service. While 6.1.10 contains the .net plumbing for a windows service, this project contains the implementation. As per the comments in the single class in this assembly, instantiating the class and holding a reference to it runs the service.

The implementation is split into this separate assembly so that it can be tested from 6.1.12.

### NQueue.Service.Test

A console application that can be used to host and run\test the batch processing service without installing as a windows service – it's easier to debug than a windows service.

### NQueue.UnitTests

Unit tests for the batch framework.

### NQueueCmd

Console application that allows external processes to enqueue work items for immediate execution. Syntax is

NQueuecmd.exe *JobInstanceName [ServerAffinity]*

Where:

*JobInstanceName* is the name of the job instance to enqueue. The name must be exactly as configured within the system.

*ServerAffinity* is an optional parameter specifying if the job should run on a specific server. If omitted the job will run on any server running instances of the host the job instance is assigned to (see Figure 2: Software Architecture). If specified the server must be configured to run an instance for this job instance’s host. The shortcut “(local)” can be used to specify the local servername to run the job on “this server only”.

### NQueue\_BatchAdmin.Setup

Setup project for the batch monitor website

### NQueue.Setup

Setup project for the main tool and binaries

1. The provided script *CleanupDatabase.sql* can be run to wipe all configuration and return the database to a vanilla state. [↑](#footnote-ref-1)
2. This can cause unexpected effects at the boundaries of processing. For example; if at 3pm you create a new schedule for a job instance that runs daily at 4pm, the job will likely not run *today* at 4pm as the daily schedule type for today has probably already been evaluated (if any host instances are already running, the schedule type was probably evaluated a few seconds past midnight). [↑](#footnote-ref-2)
3. Once a minute [↑](#footnote-ref-3)