****

Processing framework

If | Hades

Findwise Stockholm

2012-09-26

Table of Contents

Introduction 4

Input service execution sequence 4

Jobs runner execution sequence 4

AdminUI 5

Findwise.Hades.CastleIntegration 5

Core 6

ApplicationServices 6

Data 6

Dispatch 6

Input 6

JobsRunner 6

QuartzIntegration 7

Python 7

Tests 8

Admin UI 9

Global status 9

Domain actions 9

Search bar 10

Domain details 11

System status 11

Deployment 13

Prerequisites 13

Deployment 13

Running as an console application 13

Running as an windows service 13

Configuration 14

Input service 14

Jobs runner 14

App.config 14

quartz\_jobs.xml 14

Admin UI 15

Technologies 15

Windows Communication Foundation (WCF) 15

Windsor Castle 16

Quartz Net 16

Topshelf 16

Log4Net 16

MongoDb 16

ASP.NET MVC 3.0 16

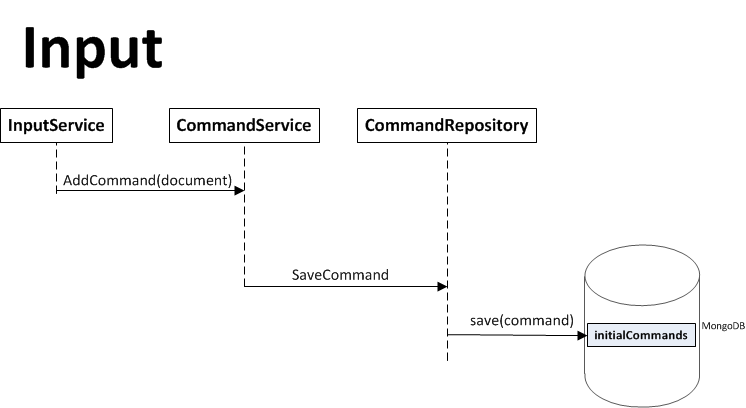
# Introduction

Hades is a processing framework featuring a document persistence store and using scripts for processing documents. The system consists of three separate applications:

* **Input service** A windows service exposing endpoints with an interface that enables consumers to send documents to it.
* **Jobs runner**  C:\Users\magnus.mansson\Downloads\Hades.Runner.pngA windows service that hosts scheduled jobs. Jobs that enables concurrent processing and dispatching of documents.
* **Admin UI**An MVC.Net website that allows an administrator to get an overview of all stored documents and control the processing/dispatching flow, as well as re-indexing tasks.

# Input service execution sequence

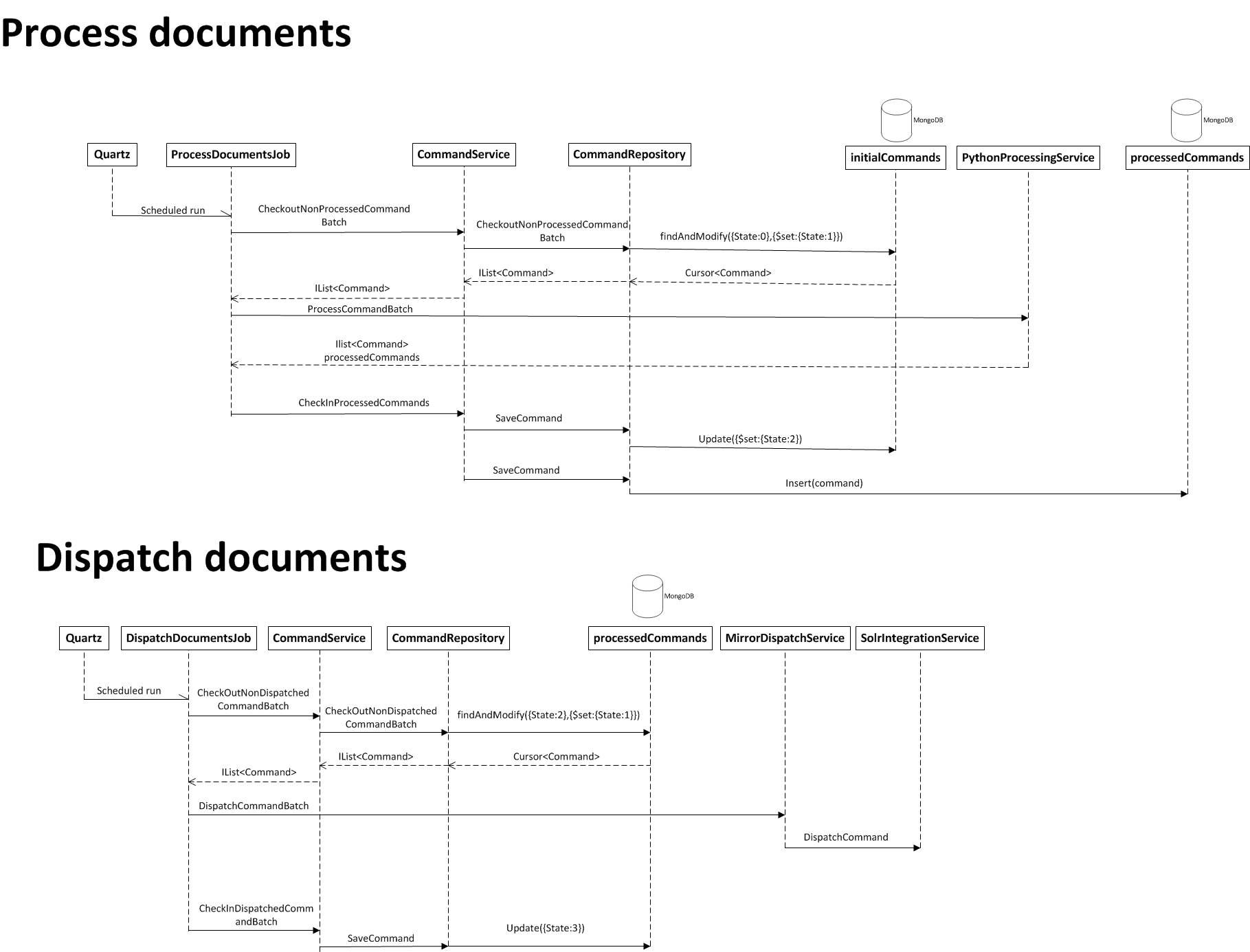
The input service’s main functionality is to receive documents and persist them in a configured persistence store (mongoDB so far). Below is a diagram of the execution sequence when a document is added and what classes that are involved in the persistence work.



Jobs runner execution sequence

This application’s main functionality is to run scheduled jobs. A job is represented by a class (within the QuartzIntegration project) that can be configured using xml in the quartz\_jobs.xml file. The two most crucial jobs for the indexing to work are the **ProcessJob** and the **DispatchJob.** These two will, in an isolated manner, process a set of documents and dispatch them to the configured integration points (Solr).

Below is an execution sequence of all the classes and their functions that are involved when the two jobs are executed.

****

### AdminUI

The AdminUI for Hades presents a simple web interface to an admin for monitoring processing and indexing status of hades. Requires ASP.NET MVC 3.0 installed on the IIS in order to run.

### Findwise.Hades.CastleIntegration

The purpose of the CastleIntegration project is to wire the whole app together. The project mainly contains installers for various components and most notably the Bootstrapper.cs which encapsulates the installers into two packages: one for the Jobs Runner’s end-project and one for the Input end-project.

If you implement a new service or create a dependency on some new instance of a class this is where you should register the required instance for the application to become aware of it.

### Core

The core project is divided into a couple of main parts

### ApplicationServices

This folder contains all services that will be running in any deploy of Hades. Most notably we have the ICommandService which handles any and all input and output of Commands. They are retrieved through this service as well as updated by use of this service. Its purpose is to provide Commands for processing and Dispatching as well as creating new Commands from an input Document.

#### Configuration

This is a heap of utility functions used by various classes to programmatically use the .Net xml configuration files. These are typically never changed unless a bug is found.

### Data

This folder currently only houses Command repositories. For simplicity there is an IInitialCommandRepository as well as an IProcessedCommandRepository which has all of its functionality derived from ICommandRepository.

We have currently implementations towards MongoDB and RavenDB for these interfaces.

### Dispatch

The dispatch project handles dispatching of documents. This is realized in implementations of the two interfaces IDispatchService and IIntegrationService. The **IDispatchService** handles strategies for dispatching a set of Commands. The implementation will typically have a dependency on an implementation of **IIntegrationService** which provides functionality for sending an add and a delete to a specific search engine.

### Input

This is one of the runnable end-projects of the solution. The input project exposes an endpoint defined by **IInputService** which has a REST interface for sending documents as well as deletes to Hades. The difference between an add and a delete is conceptually that a delete only knows the id of the document to be deleted. Therefore the delete calls expect a list of type string, i.e. the ids, while the add calls expect a Document.

As a help and further documentation of what the calls should look like the IInputService exposes example requests on the form “ExampleX” where X is the operation. If new functionality is added to IInputService it is common praxis to also add an example which specifies a syntactically correct way to make a call to the new functionality.

All de-serialization of objects is handled by System.Runtime.Serialization.

### JobsRunner

The second runnable end-project is the one which handles processing and dispatching of documents to the search engine. It uses QuartzIntegration to execute “jobs” defined in the quartz\_jobs.xml. This is also where processing scripts used by the Python processing service is stored in its Scripts folder.

### QuartzIntegration

Hades uses Quartz for scheduling and execution of “jobs”. The jobs are defined in the “Jobs” folder and have implementation of ConcurrentJobBase for each job to be scheduled. Currently there are three jobs implemented for **Dispatch**, **Processing** and **ResetErrors**. They are all configured similarly in the quartz\_jobs.xml file, here is an example:

<job>

<name>ProcessDocuments</name>

<group>DEFAULT</group>

<description>Process a batch of documents</description>

<job-type>Findwise.Hades.QuartzIntegration.Jobs.ProcessDocumentsJob, Findwise.Hades.QuartzIntegration</job-type>

<durable>true</durable>

<recover>false</recover>

<job-data-map>

<entry>

<key>ConcurrentJobLimit</key>

<value>1</value>

</entry>

<entry>

<key>BatchSize</key>

<value>1</value>

</entry>

</job-data-map>

</job>

This xml entry defines a job called ProcessDocuments. Notable here is the two configuration parameters ConcurrentJobLimit and BatchSize which regulate how the job will be executed. ConcurrentJobLimit tells ConcurrentJobBase that this is how many concurrent executions of the job that will be ran on each trigger. Set this value to -1 to indicate that the job should never be run. A suggested value of ConcurrentJobLimit is 3-5 which means that 3-5 threads will be spawned on each trigger of the job.

<trigger>

<cron>

<name>ProcessDocuments-trigger</name>

<group>DEFAULT</group>

<job-name>ProcessDocuments</job-name>

<job-group>DEFAULT</job-group>

<misfire-instruction>DoNothing</misfire-instruction>

<cron-expression>\* \* \* \* \* ?</cron-expression>

</cron>

</trigger>

For each job requires a defined trigger for it if it is to be ran at any point in time. The trigger references a specific job through its <job-name> xml field. The actual schedule is defined by the cron expression, more information about how to write those can be found here: <http://en.wikipedia.org/wiki/Cron#Format>.

## Python

The python project has a python implementation of IDocumentProcessingService which is used by the ProcessDocumentsJob. It basically examines for each Command its domain and looks for a script in the Scripts folder named [domain]Script.py and tries to run ProcessDocument(command.Document.Clone()) or ProcessDelete(command.Document.Clone()).

The python scripts have access to a couple of utility functions. For example it is common praxis to wrap all definitions of ProcessDocument and ProcessDelete with a try: catch: handleException(input) which ensures that if the script fails for some reason the output document will hold information from python about the crash rather than a generic C# exception. This is captured by adding a Field to the document with the name “PythonError” and the stacktrace as the value. The PythonProcessingService checks any Document returned by a script for existence of this field and writes a proper Error instance to the relevant command.

More advanced processing will need utility functions written in C#. If they are specific for the source it is usually captured in a static C# class called XUtil where X is the source name.

#### IronPython

We use IronPython for integration with C#. Here are some handy notes about that integration.

To load a C# class from IronPython, run with ipy.exe for example, is done by first importing clr which is an IronPython specific python module. Using clr it is possible to load assemblies with clr.AddReference(assembyName).

Conversely to load C# classes into IronPython from C#, which is what’s being done in PythonProcessingService, you need to add the Assembly object to the python runtime. This is done in the LoadAssembliesIntoPython method of PythonProcessingService.

## Tests

All tests that exist for Hades are located in the “Tests” project.

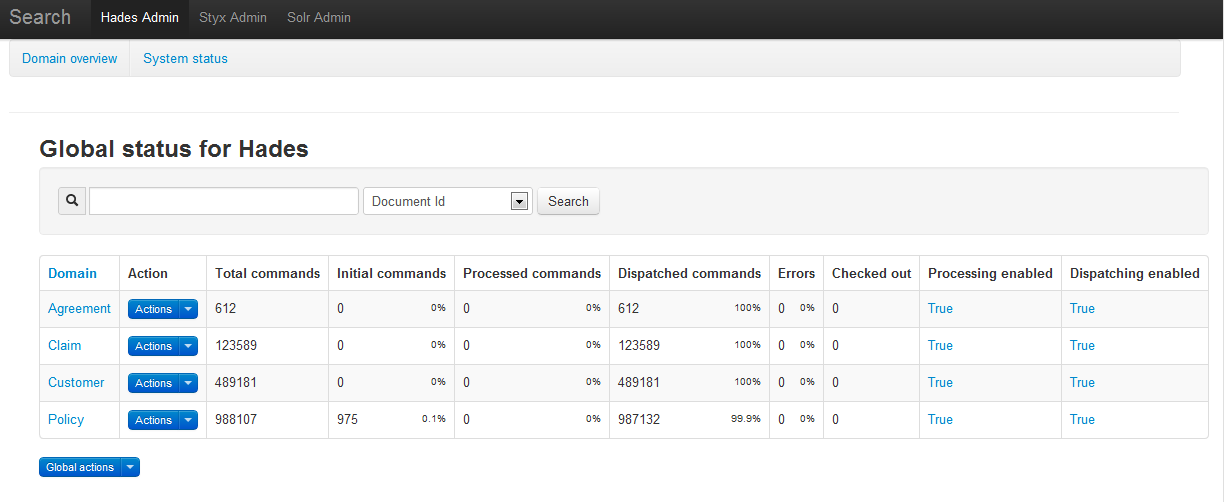
# Admin UI

The administration interface can be used to get a good overview of the status of Hades and all documents in it. Processing as well as dispatching of documents can be controlled using the different controls found.

## Global status

This is the main page that will give you an overview of all existing domains’ status. How many received, processed and dispatched documents that exist. You can follow the jobs runner’s work here by refreshing the page and see commands moving from one status to “Checked out” and then advancing to the next status (initial -> checked out -> processed -> checked out -> dispatched). If an error occurs in any of the stages the command’s state will be set to “Error”.

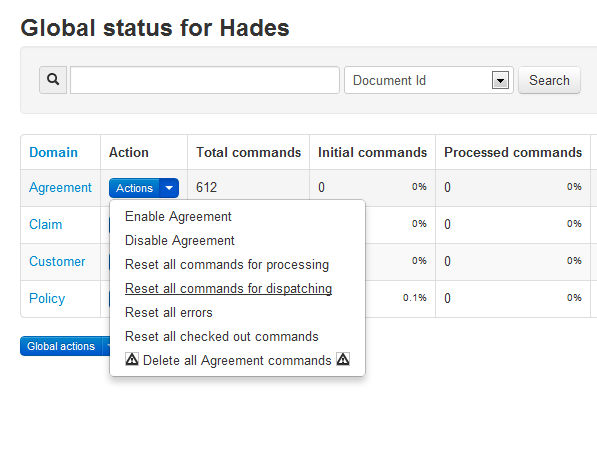
The column to the far right shows if processing or/and dispatching is enabled. Clicking on any of these states will change it

Clicking on the domain name will take you to the domain overview page where you can see the latest commands and find more information about them.

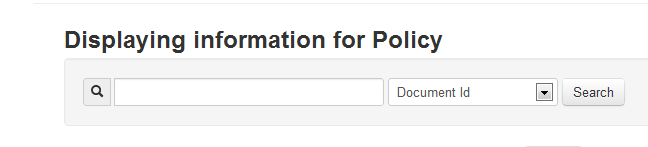
### Domain actions

If you press the arrow I the “action” button a drop down list with several actions will be presented. Beware that you don’t accidently click any of these since some actions can take quite some time.

* **Enable [Domain]**  
  Will set this domain’s processing and dispatching flags to true
* **Disable [Domain]**  
  Will set tis domain’s processing and dispatching flags to false
* **Reset all commands for processing**  
  Will set all commands to their initial state. Any processed documents will be deleted from the persistence store.
* **Reset all commands for dispatching**  
  Will set all dispatched commands to processed, enabling the DispatchJob job to pick these commands up again and dispatch them to the configured index
* Reset all errors  
  Any commands in Error state will be set to initial
* **Reset all checked out commands**  
  Any commands that are in a checked out state will be set to initial. This can be useful to do if you find that some commands never seem to “come out of” their checked out command. The cause can be that a process was terminated during processing/dispatching
* **Delete all [domain] commands**  
  All commands that are stored (both initial and processed) will be deleted from Hades.



## Search bar

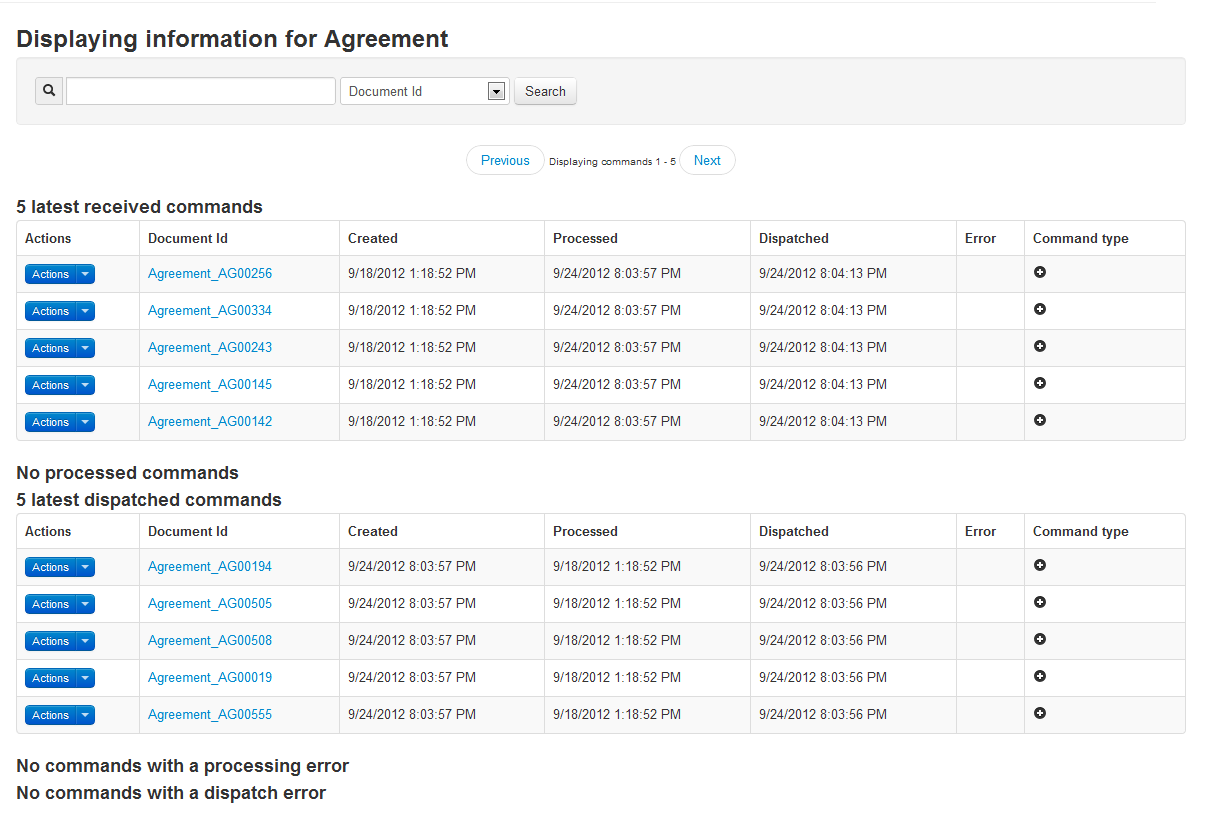
Using the search bar that is deployed in the top of the interface enables you to search for a specific document. Using the document’s exact id you will find out if that document exists in Hades when clicking “Search”. In the result list click the document to see more detailed information about it. 

## Domain details

Clicking on any of the domain names in the global status page will take you to a new page with more detailed information about the domain you clicked. Here the latest received, processed, dispatched and erroneous commands will be listed, as well as information on when they were received and what type they are. Command type can either be add (plus sign) or delete (minus sign). An add command means that the command has a document that will be added when it’s dispatched. A delete means that it contains an Id of a document that will be deleted once it’s dispatched to the index.

This view also allows you to reset individual commands for processing, using the actions button shown in each command row.

On top of this page, just below the search bar, there’s a pagination control that allow you to show the next 5 commands for each table below.

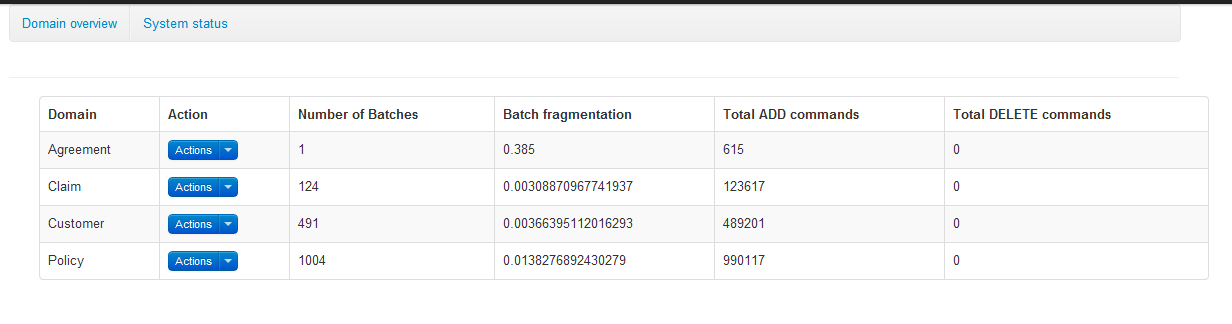


## System status

This page will show more technical details about Hades and the underlying persistence store. There are two actions that can be executed for each domain: “Optimize” and “Repair”.

Optimize will try to merge batches so that existing small batches are merged and fewer but larger batches (still conforming to the BatchLimit setting) are created. Look at the “Batch Fragmentation” value to see if this is necessary. If it’s above 0.5 this means that there are a lot of small batches that should be merged. Note that this action will take quite some time to execute.

The repair action will verify the batch logic’s integrity and any irregularities will be fixed. Unfortunately errors will occur in concurrent, multi-threaded systems. Use this if there seems to be commands that always gets stuck in Checked out state or that never are processed at all.



# Deployment

## Prerequisites

* Hades requires .Net 4.0 installed on their hosting environments.
* Hades relies on a MongoDb instance for document caching, states management and administration. For Hades to start it needs to be able to connect to the MongoDB instance.
* The MongoDb intance will be heavily loaded during processing.
* The Hades admin UI requires that ASP MVC 3.0 is installed for on the hosting server.
* Preferred, but not required for Hades to start, is that Hades has access to a running a Solr instance.
* The machine that Admin UI is deployed at needs to have MVC.Net 3 installed (you might need to reset the IIS). You also might need to register Asp.Net 4 in the IIS. To do this run the following command (x86 and x64 systems):
  + %windir%\Microsoft.NET\Framework\v4.0.30319\aspnet\_regiis.exe –ir
  + %windir%\Microsoft.NET\Framework64\v4.0.30319\aspnet\_regiis.exe -ir

## Deployment

The two executables, **Findwise.Hades.Input.exe** and **Findwise.Hades.JobsRunner.exe**,can be run as standalone a console application or as a windows service. **Note** that the executing user must have privileges to expose endpoints (open a port) or else the application will fail to start.

**Findwise.Hades.Input**

Copy the content from %deployment build% into your deployment folder for the Input service

**Findwise.Hades.JobsRunner**

Copy the content from %deployment build% into your deployment folder for the Hades Jobsrunner.

**Hades.AdminUI**

Copy the content from %deployment build% into your deployment folder for AdminUI. Then go to IIS manager and deploy the application as an IIS-site. Make sure you turn Windows security on if you need it to be secured. The allowed users/groups can be configured using either the IIS interface or modifying the web.config directly.

## Running as an console application

Double click the executable (you might need to right click the file and *run as administrator*). All output can be seen in the console window.

## Running as an windows service

Execute **Findwise.Hades.[*Input****/****Jobsrunner*].exe install** from command line. If the installation was successful you will see a new service (default Hades.JobsRunner and Hades.Input, can be changed in configuration file) window’s service manager.

# Configuration

## Input service

The input service is configured in the app.config file located in the root of the application folder. The notable configuration parameters are

* **MongoDbUrl**  
  The URL to the mongodb database e.g. *mongodb://localhost/HadesStore*
* **CommandBatchSize**   
  Specify the maximum size of a command batch. Incoming document are logically grouped into batches that as most are this size. If you have a lot of big documents this value should be tuned down a bit.  
  Recommended value is **1000**
* **CommandHistoryEnabled**If set to true documents coming in won’t overwrite existing documents with the same id. Use this if it’s crucial to be able to follow updates and deletes. Note that this will use a lot more disk space, resulting in a bigger and slower database.  
  Recommended value is **false**
* **ServiceName**  
  The displayname that will be used when installing input service as an windows service

## Jobs runner

As with the input server Jobs runner is configured using its own app.config but it also contains configuration in *quartz\_jobs.xml* file.

### App.config

* **MongoDbUrl**   
  The url to the mongodb database for Hades (Should be same as inputserver)
* **PythonLibsLocation**  
  The directory of your IronPython library, e.g. *C:\Program Files (x86)\IronPython 2.7\libs*
* **CommandBatchSize** (see input service)
* **CommandHistoryEnabled** (see input service)
* **ServiceName**  
  The displayname that will be used when installing jobs runner as an windows service

### quartz\_jobs.xml

* Common for all jobs:
  + **ConcurrentJobLimit**  
    The maximum number of jobs of the same type that can run concurrently
* ProcessDocumentsJob
  + **BatchSizeLimit**  
    The maximum number of commands that should be processed at the same time. Note that this is only the maximum limit, not the actual number of commands that will be processed. The configured batch limit and actual batch sizes affects how many commands that are processed.
* DispatchDocumentsJob
  + **BatchSizeLimit** (see ProcessDocumentsJob)
  + **SolrIntegrationService**  
    Specify where to dispatch the documents too. Format that you should be using is a comma separated list with three parameters: [addUrl], [pingUrl], [name of id field]  
    e.g. <http://localhost:8080/solr/update>, <http://localhost:8080/solr/admin/ping>, id
* ResetErrorsJob
  + **BatchSizeLimit** (see ProcessDocumentsJob)
  + **ResetProcessErrors**If true, all processing errors will be reset to initial state when executed
  + **ResetDispatchErrors**If true, all errors that happened during dispatching phase will be reset to their processed state
* OptimizeSearchIndexJob
  + **SolrIntegrationService**  
    Address to the update endpoint where the </optimize> command will be sent

## Admin UI

The admin UI comes with a web.config containing parameters that should be set:

* **MongoDbUrl** (see above)
* **CommandBatchSize** (see above)
* If you want to add links to the top main menu you can fill in url:s for:
  + **HadesAdminUrl**
  + **StyxAdminUrl**
  + **CharonAdminUrl**
  + **SolrAdminUrl**

# Technologies

## Windows Communication Foundation (WCF)

* <http://msdn.microsoft.com/en-us/netframework/aa663324>

a part of the .NET Framework that provides a unified programming model for rapidly building service-oriented applications that communicate across the web and the enterprise.

## Windsor Castle

* <http://stw.castleproject.org/Windsor.MainPage.ashx>

The most mature inversion of control container around.

## Quartz Net

* <http://quartznet.sourceforge.net/>

Quartz is a full-featured, open source job scheduling system that can be used from smallest apps to large scale enterprise systems.

Quartz.NET is a pure .NET library written in C# and is a port of very popular open source Java job scheduling framework.

## Topshelf

* <http://topshelf-project.com/>

An easy service hosting framework for building Windows services using .NET or Mono.

## Log4Net

* <http://logging.apache.org/log4net/>

The Apache log4net library is a tool to help the programmer output log statements to a variety of output targets. log4net is a port of the Apache log4j framework.

## MongoDb

* <http://www.mongodb.org>

MongoDB is a document database in the no-sql peripheries.

## ASP.NET MVC 3.0

* <http://www.asp.net/mvc>

In order to run MVC.Net 3 applications in the IIS a separate installer needs to be downloaded (see link above) and installed.