Reference documentation
# Table of Contents

I. Introduction to DataCleaner ................................................................. 1
   1. Background and concepts .............................................................. 3
      What is data quality (DQ)? .......................................................... 3
      What is data profiling? .............................................................. 3
      What is a datastore? ................................................................. 4
      What is data monitoring? ........................................................... 4
      What is master data management (MDM)? .................................... 4
   2. Getting started with DataCleaner desktop ..................................... 5
      Installing the desktop application .............................................. 5
      Connecting to your datastore ..................................................... 5
      Adding components to the job .................................................... 6
      Wiring components together ...................................................... 7
      Executing jobs ............................................................................ 7
      Saving and opening jobs ............................................................. 8
      Template jobs ............................................................................. 8
      Writing cleansed data to files ..................................................... 10
   3. Getting started with DataCleaner monitor ...................................... 11
      Installing the monitoring web application .................................... 11
      Connecting to your datastore ..................................................... 11
      Building a job ............................................................................. 12
      Scheduling jobs .......................................................................... 13
      Adding metric charts on the dashboard ....................................... 16
   II. Analysis component reference ...................................................... 20
   4. Analyzers ....................................................................................... 22
      Completeness analyzer ..................................................................... 22
      EasyDQ matching and deduplication ........................................... 22
      Boolean analyzer .......................................................................... 23
      Character set distribution ............................................................. 23
      Date gap analyzer .......................................................................... 23
      Date/time analyzer ....................................................................... 23
      Matching analyzer ........................................................................ 23
      Number analyzer ........................................................................... 23
      Pattern finder ............................................................................... 23
      String analyzer ............................................................................. 26
      Value distribution ................................................................. 26
      Value matcher .............................................................................. 27
      Weekday distribution ................................................................... 27
   5. Transformations ............................................................................. 28
      Table lookup .................................................................................. 28
      Synonym lookup .......................................................................... 29
      JavaScript transformer ................................................................. 30
      EasyDQ services .......................................................................... 33
      Equals .......................................................................................... 34
      Max rows ..................................................................................... 35
      Not null ........................................................................................ 35
   6. Writers ............................................................................................. 36
      Create CSV file ............................................................................ 36
      Create Excel spreadsheet ............................................................. 36
      Create staging table ..................................................................... 36
      Insert into table ............................................................................ 37
   III. Reference data .............................................................................. 38
## Reference documentation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.</td>
<td>Dictionaries</td>
<td>40</td>
</tr>
<tr>
<td>8.</td>
<td>Synonyms (aka. Synonym catalogs)</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>Text file synonym catalog</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>Datastore synonym catalog</td>
<td>41</td>
</tr>
<tr>
<td>9.</td>
<td>String patterns</td>
<td>42</td>
</tr>
<tr>
<td>IV.</td>
<td>Configuration reference</td>
<td>43</td>
</tr>
<tr>
<td>10.</td>
<td>Configuration file</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>XML schema</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Datastores</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Database (JDBC) connections</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Comma-Separated Values (CSV) files</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Fixed width value files</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Excel spreadsheets</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>XML file datastores</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>MongoDB databases</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>CouchDB databases</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Reference data</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Task runner</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Storage provider</td>
<td>50</td>
</tr>
<tr>
<td>11.</td>
<td>Analysis job files</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>XML schema</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Source section</td>
<td>52</td>
</tr>
<tr>
<td>12.</td>
<td>Logging</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Logging configuration file</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Default logging configuration</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Modifying logging levels</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>Alternative logging outputs</td>
<td>56</td>
</tr>
<tr>
<td>V.</td>
<td>DataCleaner monitor repository</td>
<td>57</td>
</tr>
<tr>
<td>13.</td>
<td>Repository configuration</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Configure repository location</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Providing signed Java WebStart client files</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Cluster configuration (distributed execution)</td>
<td>60</td>
</tr>
<tr>
<td>14.</td>
<td>Repository layout</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Multi-tenant layout</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Tenant home layout</td>
<td>62</td>
</tr>
<tr>
<td>VI.</td>
<td>DataCleaner monitor web services</td>
<td>63</td>
</tr>
<tr>
<td>15.</td>
<td>Job triggering</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Trigger service</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Polling for execution status</td>
<td>66</td>
</tr>
<tr>
<td>16.</td>
<td>Repository navigation</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Job files</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Result files</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Uploading content to the repository</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>Modifying result metadata</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>Renaming jobs</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>Copying jobs</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>Deleting jobs</td>
<td>70</td>
</tr>
<tr>
<td>17.</td>
<td>Metric web services</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>Metrics background</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>Getting a list of available metrics</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>Getting the values of particular metrics</td>
<td>71</td>
</tr>
<tr>
<td>18.</td>
<td>Atomic transformations (data cleaning as a service)</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>What are atomic transformation services?</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Invoking atomic transformations</td>
<td>73</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>VII. Invoking DataCleaner jobs from the command-line</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>19. Command-line interface</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>Executables</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>Usage scenarios</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>Executing an analysis job</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>Listing datastore contents and available components</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>Parameterizable jobs</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>Dynamically overriding configuration elements</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>VIII. Third party integrations</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>20. Pentaho integration</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>Launch DataCleaner to profile Pentaho Data Integration steps</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>Run Pentaho Data Integration jobs in DataCleaner monitor</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>Run DataCleaner jobs in Pentaho Data Integration</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>IX. Developer's guide</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>21. Architecture</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Data access</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Processing framework</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>22. Executing jobs through code</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>Overview of steps and options</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>Step 1: Configuration</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>Step 2: Job</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>Step 3: Execution</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Step 4: Result</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>23. Developer resources</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>Extension development tutorials</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>Building DataCleaner</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>24. Extension packaging</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>Annotated components</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>Single JAR file</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>Extension metadata XML</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>Component icons</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>25. Embedding DataCleaner</td>
<td>106</td>
<td></td>
</tr>
</tbody>
</table>
## List of Tables

4.1. Pattern finder properties ................................................................. 24  
4.2. Value distribution properties ............................................................ 26  
5.1. JavaScript variables ........................................................................... 31  
5.2. JavaScript data types .......................................................................... 32  
15.1. Job triggering HTTP parameters .......................................................... 65
Part I. Introduction to DataCleaner
# Table of Contents

1. Background and concepts ................................................................. 3  
   What is data quality (DQ)? ............................................................... 3  
   What is data profiling? ................................................................. 3  
   What is a datastore? ........................................................................ 4  
   What is data monitoring? ............................................................... 4  
   What is master data management (MDM)? ...................................... 4  

2. Getting started with DataCleaner desktop ........................................ 5  
   Installing the desktop application .................................................. 5  
   Connecting to your datastore ......................................................... 5  
   Adding components to the job ......................................................... 6  
   Wiring components together ......................................................... 7  
   Executing jobs .............................................................................. 7  
   Saving and opening jobs ............................................................... 8  
   Template jobs .............................................................................. 8  
   Writing cleansed data to files ....................................................... 10  

3. Getting started with DataCleaner monitor ....................................... 11  
   Installing the monitoring web application ..................................... 11  
   Connecting to your datastore ....................................................... 11  
   Building a job ............................................................................. 12  
   Scheduling jobs ........................................................................... 13  
   Adding metric charts on the dashboard ........................................ 16
Chapter 1. Background and concepts

Abstract

In this chapter we will try to define how we see the concepts and terms surrounding the environment(s) around DataCleaner.

Although these terms have no strict definitions, you can use this chapter as a guide, at least for the scope of how to use and what to expect from DataCleaner in relation to the described topics.

As a lot of the statements in this chapter are in deed subjective or based upon personal experience, we encourage everyone to provide their feedback and to contribute corrections/improvements to it.

What is data quality (DQ)?

Data Quality (DQ) is a concept and a business term covering the quality of the data used for a particular purpose. Often times the DQ term is applied to the quality of data used in business decisions but it may also refer to the quality of data used in research, campaigns, processes and more.

Working with Data Quality typically varies a lot from project to project, just as the issues in the quality of data vary a lot. Examples of data quality issues include:

1. Completeness of data
2. Correctness of data
3. Duplication of data
4. Uniformedness/standardization of data

A less technical definition of high-quality data is, that data are of high quality "if they are fit for their intended uses in operations, decision making and planning" (J. M. Juran).

A data quality analysis (DQA) is the (human) process of examining the quality of data for a particular process or organization. The DQA includes both technical and non-technical elements. For example, to do a good DQA you will probably need to talk to users, business people, partner organizations and maybe customers. This is needed to assess what the goal of the DQA should be.

From a technical viewpoint the main task in a DQA is the data profiling activity, which will help you discover and measure the current state of affairs in the data.

What is data profiling?

Data profiling is the activity of investigating a datastore to create a 'profile' of it. With a profile of your datastore you will be a lot better equipped to actually use and improve it.

The way you do profiling often depends on whether you already have some ideas about the quality of the data or if you're not experienced with the datastore at hand. Either way we recommend an explorative approach, because even though you think there are only a certain amount of issues you need to look for, it is our experience (and reasoning behind a lot of the features of DataCleaner) that it is just as important to check those items in the data that you think are correct! Typically it's cheap to include a bit more data into your analysis and the results just might surprise you and save you time!
DataCleaner comprises (amongst other aspects) a desktop application for doing data profiling on just about any kind of datastore.

**What is a datastore?**

A datastore is the place where data is stored. Usually enterprise data lives in relational databases, but there are numerous exceptions to that rule.

To comprehend different sources of data, such as databases, spreadsheets, XML files and even standard business applications, we employ the umbrella term *datastore*. DataCleaner is capable of retrieving data from a very wide range of datastores. And furthermore, DataCleaner can update the data of most of these datastores as well.

**What is data monitoring?**

We've argued that data profiling is ideally an explorative activity. Data monitoring typically isn't! The measurements that you do when profiling often times needs to be continuously checked so that your improvements are enforced through time. This is what data monitoring is typically about.

Data monitoring solutions comes in different shapes and sizes. You can set up your own bulk of scheduled jobs that run every night. You can build alerts around it that send you emails if a particular measure goes beyond it's allowed thresholds, or in some cases you can attempt ruling out the issue entirely by applying First-Time-Right (FTR) principles that validate data at entry-time. eg. at data registration forms and more.

As of version 3, DataCleaner now also includes a monitoring web application, dubbed "DataCleaner monitor". The monitor is a server application that supports orchestrating and scheduling of jobs, as well as exposing metrics through web services and through interactive timelines and reports. It also supports the configuration and job-building process through wizards and management pages for all the components of the solution. As such, we like to say that the DataCleaner monitor provides a is a good foundation for the infrastructure needed in a Master Data Management hub.

**What is master data management (MDM)?**

Master data management (MDM) is a very broad term and is seen materialized in a variety of ways. For the scope of this document it serves more as a context of data quality than an activity that we actually target with DataCleaner per-se.

The overall goals of MDM is to manage the important data of an organization. By "master data" we refer to "a single version of the truth", ie. not the data of a particular system, but for example all the customer data or product data of a company. Usually this data is dispersed over multiple datastores, so an important part of MDM is the process of unifying the data into a single model.

Obviously another of the very important issues to handle in MDM is the quality of data. If you simply gather eg. "all customer data" from all systems in an organization, you will most likely see a lot of data quality issues. There will be a lot of duplicate entries, there will be variances in the way that customer data is filled, there will be different identifiers and even different levels of granularity for defining "what is a customer?". In the context of MDM, DataCleaner can serve as the engine to cleanse, transform and unify data from multiple datastores into the single view of the master data.
Chapter 2. Getting started with DataCleaner desktop

Installing the desktop application

The desktop version of DataCleaner requires practically no installation. The application is not dependent on any particular operating system and does not need to "register" in any registration database or something like that.

The only two requirements of DataCleaner are:

1. A computer (with a graphical display, except if run in command-line mode).

You have two options in terms of installation.

1. Download DataCleaner as a distributable package (.zip or .tar.gz) from the release list on our downloads page [http://datacleaner.org/downloads].

   Unpackage the distributable in a directory of your own choice. DataCleaner will save its configuration within this same directory.

2. Run DataCleaner as a Java WebStart application. You can do this by clicking the WebStart link on our downloads page [http://datacleaner.org/downloads].

   In WebStart the application will be installed in your JRE's application cache. Each time you start the application it will automatically check for updates to the application. DataCleaner's configuration will be stored in your "home"-directory in a folder called .datacleaner/[version].

Connecting to your datastore

Below is a screenshot of the initial screen that will be presented when launching DataCleaner (desktop edition), containing a list of datastores. Above the list you will see a set of icons, each representing a type of datastore. Click either of the icons to register your own datastore.
Once you've registered ('created') your own datastore, you can select it from the list and click 'Analyze!' to start working with it!

**Tip**

You can also configure your datastore by means of the configuration file (conf.xml), which has both some pros and some cons. For more information, read the configuration file chapter.

**Adding components to the job**

There are a few different kinds of components that you can add to your job:

1. **Analyzers**, which are the most important components. Actually, without at least one analyzer the job will not run. An analyzer is a component that inspects the data that it receives and generates a result or a report. The majority of the data profiling cruft is created as analyzers.

2. **Transformers** are components used to modify the data before analyzing it. Sometimes it's necessary to extract parts of a value or combine two values to correctly get an idea about a particular measure. In other scenarios, transformers can be used to perform reference data lookups or other similar tasks and place the results of an operation into the stream of data in the job.

   The result of a transformer is a set of output columns. These columns work exactly like regular columns in your job, except that they have a preceeding step in the flow before they become materialized.

3. **Filters** are components that split the flow of processing in a job. A filter will have a number of possible outcomes and depending on the outcome of a filter, a particular row might be processed by different sub-flows. Filters are often used simply to disregard certain rows from the analysis, eg. null values or values outside the range of interest.

   Each of these components will get their own tab, from where you can configure them.
Transformers and filters are added to your job using the "Transform" button. Please refer to the reference chapter Transformations for more information on specific transformers and filters.

Analyzers are added to your job using the "Analyze” button. Please refer to the reference chapter Analyzers for more information on specific analyzers.

## Wiring components together

Simply adding a transformer or filter actually doesn't change your job as such! This is because these components only have an impact if you wire them together somehow.

To wire a transformer you simply need to use it's output column. DataCleaner will automatically build the flow so that a transformer is executed before components that depend on it's output columns.

To wire a filter you need to set up a dependency on either of it's outcomes. All components have a button for selecting filter outcomes in their top-right corners. Click this button to select a filter outcome to depend on. If you have multiple filters you can chain these simply by having dependent outcomes of the individual filters.

To get an overview of your current job flow, you can click the "Visualize” button, which will present the job's contents in an interactive flow diagram:

![Flow Diagram](image)

## Executing jobs

When a job has been built you can execute it. To check whether your job is correctly configured and ready to execute, check the status bar in the bottom of the job building window.

To execute the job, simply click the "Run analysis" button in the top-right corner of the window. This will bring up the result window, which contains:

1. The *Progress information* tab which contains useful information and progress indications while the job is executing.

   2. Additional tabs for each table that is being processed in the job. Results for the individual analyzers will be shown in these tabs.
Here's an example of an analysis result window:

![Analysis Result Window](image)

### Saving and opening jobs

You can save your jobs in order to reuse them at a later time. Saving a job is simple: Simply click the "Save analysis job" button in the top-left corner of the window.

Analysis jobs are saved in files with the "analysis.xml" extension. These files are XML files that are readable and editable using any XML editor.

Opening jobs can be done using the "File -> Open analysis job..." menu item. Opening a job will restore a job building window from where you can edit and run the job.

### Template jobs

DataCleaner contains a feature where you can reuse a job for multiple datastores or just multiple columns in the same datastore. We call this feature 'template jobs'.

When opening a job you are presented with a file chooser. When you select a job file a panel will appear, containing some information about the job as well as available actions:
If you click the 'Open as template' button you will be presented with a dialog where you can map the job's original columns to a new set of columns:
Getting started with DataCleaner desktop

First you need to specify the datastore to use. On the left side you see the name of the original datastore, but the job is not restricted to use only this datastore. Select a datastore from the list and the fields below for the columns will become active.

Then you need to map individual columns. If you have two datastore that have the same column names, you can click the "Map automatically" button and they will be automatically assigned. Otherwise you need to map the columns from the new datastore's available columns.

Finally your job may contain 'Job-level variables'. These are configurable properties of the job that you might also want to fill out.

Once these 2-3 steps have been completed, click the "Open job" button, and DataCleaner will be ready for executing the job on a new set of columns!

Writing cleansed data to files

Although the primary focus of DataCleaner is analysis, often during such analysis you will find yourself actually improving data by means of applying transformers and filters on it. When this is the case, obviously you will want to export the improved/cleansed data so you can utilize it in other situations than the analysis.

Please refer to the reference chapter Writers for more information on writing cleansed data.
Chapter 3. Getting started with DataCleaner monitor

Installing the monitoring web application

In addition to (and in some cases, even as a replacement for) the desktop version of DataCleaner, we also provide a web application for monitoring, scheduling and sharing analysis jobs and results.

A Java servlet container and web server is required to run the monitoring web application. An example of this is Apache Tomcat 7.x [http://tomcat.apache.org/download-70.cgi], which is often used and tested by the DataCleaner development team.

To install the monitoring web application, download [http://datacleaner.org/downloads] the Web Archive (.war) distribution of DataCleaner. Install the .war file in your container. If you’re using Apache Tomcat, this is done by copying the .war file to the "webapps" folder within your tomcat directory. Afterwards, start the container and go to http://localhost:8080/DataCleaner-monitor to see the welcome/login screen:

In the community edition of DataCleaner, you will find suggestions for login credentials directly on the screen, to get you started quickly.

Connecting to your datastore

DataCleaner monitor displays and manages datastores through the 'Datastores' page. Click the "Register datastore" button to set up a new datastore:
Follow the on-screen instructions to go through the process of registering the datastore. When the wizard is finished you can find the datastore on the 'Datastores' page where you can work with it in different ways - e.g. do ad-hoc querying, download the data (if it is a file-based datastore), launch DataCleaner desktop to build custom jobs etc.

**Tip**

You can also configure your datastore by means of the configuration file (conf.xml), which has both some pros and some cons. For more information, read the configuration file chapter.

**Building a job**

There are multiple ways to add a new job to DataCleaner monitor. The most fail-safe way is to use one of the job wizards. These are found on the 'Scheduling' page, using the "Build job" button.
Getting started with DataCleaner monitor

Simply follow the on-screen instructions to build the job according to the wizard's intent.

Once the job has been built, you will have options to trigger its execution, schedule it, set up alerting and start building monitoring timelines for its result's metrics.

Alternatively, you can create the job using the desktop application. In that case, make sure that the name of any involved datastore in the desktop application matches the name of the datastore in the repository! If so, you can safely drop the .analysis.xml job file in the jobs folder on the repository. It will be immediately available in the web UI of the monitor application.

**Scheduling jobs**

The DataCleaner monitor web application is the preferred application for scheduling of DataCleaner jobs.

Below is a screenshot of the Scheduling page of the DataCleaner monitor. To access this page, your user needs to have the ROLE_SCHEDULE_EDITOR security role.
Getting started with DataCleaner monitor

Things to notice on this page:

1. Each job is categorized by their datastores. This is to provide an overview of which jobs are executing based on which datastores. Technically, a job can interact with more than one datastore, but it will be categorized on this page based on which datastore is acting as source for the job's execution flow.

2. Besides the datastore and job name, we have a Scheduling column. In the case shown, all jobs are 'Manually triggered', meaning that they are not scheduled by the DataCleaner monitor. Click the link to change the trigger:
Getting started with
DataCleaner monitor

The three types of triggering is explained on the screen: Periodic trigger, Dependent trigger, Manual trigger.

3. In the next column Alerts are defined. Alerts define conditions that should be inspected after execution. An alert is based on metrics, similar to those that can be shown in the timelines on the Dashboard page. An alert also contains a severity, defining how severe the alert is, if it is raised. The severity scale is (increasing severity): Intelligence, Surveillance, Warning, Fatal.

   Typical handling of raised alerts include sending an email or SMS to the data stewards. These alert notification mechanisms are included in the commercial editions of DataCleaner. You can also develop your own alert notification handlers, by implementing the org.eobjects.datacleaner.monitor.alertnotification.AlertNotifier interface in Java.

4. In the last column, Actions can be found. The following options are there:
a. *Trigger*, which triggers a job for immediate execution.

b. *Open*, which launches the DataCleaner desktop application with the job loaded. Note that for security and bandwidth concerns, only the first 20 input records of the job is transferred to the desktop application. It is in other words not possible to "trigger" the job run on the desktop as if it is running on the server, but it provides a good playground for experiments and testing. Jobs can even be modified and saved back onto the monitoring server this way.

c. *History*, which displays a dialog with execution log information about each triggered job execution.

### Adding metric charts on the dashboard

On the 'Dashboard' page you can set up a visual representation of the metrics that your jobs are gathering. The DataCleaner monitor comes with some example timeline charts that demonstrate how metrics are plotted, that you can click at a point in these timeline charts and see further details about those metrics.

Now let's see how we can define our own timelines. Select the appropriate group on the left-hand side of the page and then click "New timeline chart". You'll have to select the job which to base the metric data upon and then select from the metrics made available by that job:

![Dashboard screenshot](image)

Note that the contents of the dialog is highly dependent on the contents of the selected job, so the screenshot above will not necessarily resemble the options available to you.

There are four different types of metrics in DataCleaner monitor, and they may or may not be exposed through your job:
1. **Non-parameterized metrics.** These are the most "plain" metrics, which will be represented only by a checkbox.

2. **String-parameterized metrics.** These are metrics that take a string parameter. For instance we see that the "Match count" metric of the "Pattern finder" component takes a string parameter (screenshot above). In this case the parameter represents the pattern to look for when counting matches. Usually suggestions will be shown in the UI to help you figure out what type of parameter is expected.

3. **Column-parameterized metrics.** Some metrics are collected per column. Since we know which columns a specific component is working on, a list of columns and corresponding checkboxes are shown to make the parameter selection easier:

![String analyzer (5 columns)](image)

4. **Formula metrics.** Lastly, you can define your own metrics based on a formula of other metrics. This functionality is similar to building formulas in a spreadsheet - you simply reference other metrics and provide an equation for calculating the metric.

Adding a formula metric is done by clicking the "Add metric formula" button and filling out the formula details:
Getting started with DataCleaner monitor

Metric formulas support the basic mathematical operators: plus (+), minus (-), divide (/) and multiply (*).

When the metric selection is done, a new unsaved timeline is shown:
You can go back to selecting the metrics or customizing the chart's look and feel by clicking the 'Wrench' icon, or you can start by saving it using the save icon.

Names and colors of the metrics in the timeline chart can be customized by clicking the legend of the chart.
Part II. Analysis component reference
# Table of Contents

4. Analyzers ........................................................................................................... 22  
   Completeness analyzer ...................................................................................... 22  
   EasyDQ matching and deduplication ................................................................. 22  
   Boolean analyzer ............................................................................................... 22  
   Character set distribution .................................................................................. 23  
   Date gap analyzer .............................................................................................. 23  
   Date/time analyzer ............................................................................................ 23  
   Matching analyzer ............................................................................................. 23  
   Number analyzer .............................................................................................. 23  
   Pattern finder .................................................................................................... 23  
   String analyzer .................................................................................................. 26  
   Value distribution ............................................................................................. 26  
   Value matcher .................................................................................................... 27  
   Weekday distribution ......................................................................................... 27  
5. Transformations .................................................................................................. 28  
   Table lookup ..................................................................................................... 28  
   Synonym lookup ............................................................................................... 29  
   JavaScript transformer ...................................................................................... 30  
   EasyDQ services ............................................................................................... 33  
   Equals ................................................................................................................ 34  
   Max rows ........................................................................................................... 35  
   Not null .............................................................................................................. 35  
6. Writers .................................................................................................................. 36  
   Create CSV file ................................................................................................. 36  
   Create Excel spreadsheet .................................................................................. 36  
   Create staging table ......................................................................................... 36  
   Insert into table ................................................................................................. 37
Chapter 4. Analyzers

Abstract

This chapter deals with one of the most important concepts in DataCleaner: Analyzers. Analyzers are the endpoints of any analysis job, meaning that a job requires at least one analyzer.

An analyzer consumes a (set of) column(s) and generates an analysis result based on the values in the consumed columns.

Here is an example of a configuration panel pertaining to an analyzer:

In the panel there will always be one or more selections of columns. The configuration panel may also contain additional properties for configuration.

Completeness analyzer

The completeness analyzer provides a really simple way to check that all required fields in your records have been filled. Think of it like a big "not null" check across multiple fields. In combination with the monitoring application, this analyzer makes it easy to track which records needs additional information.

EasyDQ matching and deduplication

EasyDataQuality [http://www.easydq.com] is an on-demand service for data quality functions. DataCleaner provides access to the EasyDQ services, including the deduplication service which is used to
provide *Duplicate detection* and *Inter-Dataset matching*, but the core functionality is provided by Human Inference.

Please refer to the EasyDQ for DataCleaner documentation [http://help.easydq.com/datacleaner] for detailed information about the services provided through EasyDQ.

**Boolean analyzer**

Boolean analyzer is an analyzer targeted at boolean values. For a single boolean column it is quite simple: it will show the distribution of true/false (and optionally null) values in a column. For several columns it will also show the value combinations and the frequencies of the combinations. The combination matrix makes the Boolean analyzer a handy analyzer for use with combinations of matching transformers and other transformers that yield boolean values.

Boolean analyzer has no configuration parameters, except for the input columns.

**Character set distribution**

The Character set distribution analyzer inspects and maps text characters according to character set affinity, such as Latin, Hebrew, Cyrillic, Chinese and more.

Such analysis is convenient for getting insight into the international aspects of your data. Are you able to read and understand all your data? Will it work in your non-internationalized systems?

**Date gap analyzer**

The Date gap analyzer is used to identify gaps in recorded time series. This analyzer is useful for example if you have employee time registration systems which record FROM and TO dates. It will allow you to identify if there are unexpected gaps in the data.

**Date/time analyzer**

The Date/time analyzer provides general purpose profiling metrics for temporal column types such as DATE, TIME and TIMESTAMP columns.

**Matching analyzer**

The matching analyzer provides an easy means to match several columns against several dictionaries and/or several string patterns. The result is a matrix of match information for all columns and all matched resources.

**Number analyzer**

The number analyzer provides general purpose profiling metrics for numerical column types.

**Pattern finder**

The pattern finder is one of the more advanced, but also very popular analyzers of DataCleaner.

Here is a screenshot of the configuration panel of the Pattern finder:
From the screenshot we can see that the Pattern finder has these configuration properties:

**Table 4.1. Pattern finder properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group column</td>
<td>Allows you to define a pattern group column. With a pattern group column you can separate</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>the identified patterns into separate buckets/groups. Imagine for example that you want to check if the phone numbers of your customers are consistent. If you have an international customer based, you should then group by a country column to make sure that phone patterns identified are not matched with phone patterns from different countries.</td>
</tr>
<tr>
<td>Discriminate text case</td>
<td>Defines whether or not to discriminate (ie. consider as different pattern parts) based on text case. If true &quot;DataCleaner&quot; and &quot;datacleaner&quot; will be considered instances of different patterns, if false they will be matched within same pattern.</td>
</tr>
<tr>
<td>Discriminate negative numbers</td>
<td>When parsing numbers, this property defines if negative numbers should be discriminated from positive numbers.</td>
</tr>
<tr>
<td>Discriminate decimals</td>
<td>When parsing numbers, this property defines if decimal numbers should be discriminated from integers.</td>
</tr>
<tr>
<td>Enable mixed tokens</td>
<td>Defines whether or not to categorize tokens that contain both letters and digits as &quot;mixed&quot;, or alternatively as two separate tokens. Mixed tokens are represented using questionmark ('?') symbols.</td>
</tr>
<tr>
<td></td>
<td>This is one of the more important configuration properties. For example if mixed tokens are enabled (default), all these values will be matched against the same pattern: foo123, 123foo, foobar123, foo123bar. If mixed tokens are NOT enabled only foo123 and foobar123 will be matched (because 123foo and foo123bar represent different combinations of letter and digit tokens).</td>
</tr>
<tr>
<td>Ignore repeated spaces</td>
<td>Defines whether or not to discriminate based on amount of whitespaces.</td>
</tr>
<tr>
<td>Upper case patterns expand in size</td>
<td>Defines whether or not upper case tokens automatically &quot;expand&quot; in size. Expandability refers to whether or not the found patterns will include matches if a candidate has the same type of token, but with a different size. The default configuration for upper case characters is false (ie. ABC is not matched with ABCD).</td>
</tr>
<tr>
<td>Lower case patterns expand in size</td>
<td>Defines whether or not lower case tokens automatically &quot;expand&quot; in size. As with upper case expandability, this property refers to whether or not the found patterns will include matches if a candidate has the same type of token, but with a different size. The default configuration for lower case characters is true (ie. 'abc' is not matched with 'abc').</td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The defaults in the two &quot;expandability&quot;</td>
<td>configuration properties mean that eg. name pattern recognition is meaningful: 'James' and 'John' both pertain to the same pattern ('Aaaaa'), while 'McDonald' pertain to a different pattern ('AaAaaaaa').</td>
</tr>
<tr>
<td>Predefined token name</td>
<td>Predefined tokens make it possible to define a token to look for and classify using either just a fixed list of values or regular expressions. Typically this is used if the values contain some additional parts which you want to manually define a matching category for. The 'Predefined token name' property defines the name of such a category.</td>
</tr>
<tr>
<td>Predefined token regexes</td>
<td>Defines a number of string values and/or regular expressions which are used to match values against the (pre)defined token category.</td>
</tr>
<tr>
<td>Decimal separator</td>
<td>The decimal separator character, used when parsing numbers</td>
</tr>
<tr>
<td>Thousand separator</td>
<td>The thousand separator character, used when parsing numbers</td>
</tr>
<tr>
<td>Minus sign</td>
<td>The minus sign character, used when parsing numbers</td>
</tr>
</tbody>
</table>

## String analyzer

The string analyzer provides general purpose profiling metrics for string column types. Of special concern to the string analyzer is the amount of words, characters, special signs, diacritics and other metrics that are vital to understanding what kind of string values occur in the data.

## Value distribution

The value distribution (often also referred to as 'Frequency analysis') allows you to identify all the values of a particular column. Furthermore you can investigate which rows pertain to specific values.

Here are the configuration properties for the value distribution analyzer:

### Table 4.2. Value distribution properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group column</td>
<td>Allows you to define a column for grouping the result. With a group column you can separate the identified value distributions into separate buckets/groups. Imagine for example that you want to check if the postal codes and city names correspond or if you just want to segment your value distribution on eg. country or gender or ...</td>
</tr>
<tr>
<td>Record unique values</td>
<td>By default all unique values will be included in the result of the value distribution. This can potentially ...</td>
</tr>
</tbody>
</table>
cause memory issues if your analyzed columns contains a LOT of unique values (eg. if it's a unique key). If the actual unique values are not of interest, then uncheck this checkbox to only count (but not save for inspection) the unique values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top n most frequent vales</td>
<td>An optional number used if the analysis should only display eg. the &quot;top 5 most frequent values&quot;. The result of the analysis will only contain top/bottom n most frequent values, if this property is supplied.</td>
</tr>
<tr>
<td>Bottom n most frequent values</td>
<td>An optional number used if the analysis should only display eg. the &quot;bottom 5 most frequent values&quot;. The result of the analysis will only contain top/bottom n most frequent values, if this property is supplied.</td>
</tr>
</tbody>
</table>

**Value matcher**

The value matcher works very similar to the Value distribution, except for the fact that it takes a list of expected values and everything else is put into a group of 'unexpected values'. This division of values means a couple of things:

1. You get a built-in validation mechanism. You expect maybe only 'M' and 'F' values for your 'gender' column, and everything else is in a sense invalid, since it is unexpected.

2. The division makes it easier to monitor specific values in the data quality monitoring web application.

3. This analyzer scales much better for large datasets, since the groupings are deterministic and thus can be prepared for in the batch run.

**Weekday distribution**

The weekday distribution provides a frequency analysis for date columns, where you can easily identify which weekdays a date field represents.
Chapter 5. Transformations

Abstract

With transformations (accessible through the 'Transform' button in the main window) you can pre- and postprocess your data as part of your DQ project.

Technically speaking there are two kinds of transformations: Transformers and Filters. Transformers are used to extract, generate or refine data (new columns and sometimes also new rows), whereas filters are used to limit the dataset. In previous version (2.0 - 2.4) of DataCleaner filters and transformers were completely separated concepts, also in the user interface.

There's quite a lot of transformations available in DataCleaner, more than will be feasible to describe all in detail. This chapter provides a documentation for some of the essential ones.

Table lookup

The table lookup transformer allows you to look up values in a different table. Any amount of columns can be used for mapping (lookup conditioning) and for outputting (the outcome of the lookup).

The configuration screen for the table lookup transformer looks like this:
To make the mapping you need to select the target datastore, schema and table names. Once selected you will be able to select which columns to use for condition setting when looking up values.

The semantics of the Table lookup are close to the semantics of a LEFT JOIN. If no lookup value is found, nulls will be returned. However, if multiple records are found to match the conditions, only the first will be returned.

Note that the Table lookup will use a cache for looking up values, to avoid querying the target table for every incoming value.

**Synonym lookup**

The Synonym lookup transformation is a critical part of DataCleaner's ability to standardize and cleanse data. Using this component you can look up values in a synonym catalog and have it replaced with it's master term, if it is found to be a synonym.

Below is a screenshot of the synonym lookup's configuration panel:
The configuration of the Synonym lookup is simple:

1. Select the column to apply the lookup function to.

2. Use the 'Retain original value' option to determine if unmatched values (non-synonyms) should be retained or if a null value should be returned if there is no match.

3. Select the synonym catalog to use for lookups.

If your synonym catalog contains all the allowed values for a particular column, it can be a good idea to uncheck the 'Retain original value' checkbox and then do a simple null-check on the resulting output column. If null values are found, it’s because there are values in the column that the synonym catalog is not able to standardize.

**JavaScript transformer**

The JavaScript transformer allows the user to define his/her own script which can perform rather intricate things like conditioning, looping. It can also be used as a useful way to express small business rules.

For this documentation, a complete reference of JavaScript is out of scope. But we will show a few examples and more importantly talk about the available variables and their types.
The JavaScript transformer returns a single string. The entered script should provide this string as the last line of the script. This is why the template script is as follows (so you can just implement the eval() function):

```javascript
function eval() {
    return "hello \" + values[0];
}
eval();
```

Variables:

**Table 5.1. JavaScript variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>An array of all values in the row (as mapped by the &quot;Columns&quot; property). Using &quot;values&quot; you can reference eg. the first and third values like this:</td>
</tr>
<tr>
<td></td>
<td>var first = values[0];</td>
</tr>
<tr>
<td></td>
<td>var third = values[2];</td>
</tr>
<tr>
<td></td>
<td>Note that JavaScript arrays are 0-based. Instead of indexes you can also reference by column name, like this:</td>
</tr>
<tr>
<td></td>
<td>var idValue = values[&quot;id&quot;];</td>
</tr>
<tr>
<td>column_name</td>
<td>Any column name that is also a valid JavaScript and not a reserved variable name will also be added directly to the scope of the script as a variable. For example, if you have two columns, FIRST_NAME and LAST_NAME, you can concatenate them easily, like this:</td>
</tr>
<tr>
<td></td>
<td>var fullname = FIRST_NAME + &quot; &quot; + LAST_NAME;</td>
</tr>
<tr>
<td>out</td>
<td>A reference to the system console's &quot;out&quot; stream. If running DataCleaner with the console visible, you can print messages to the console, like this:</td>
</tr>
<tr>
<td></td>
<td>out.println(&quot;Value: &quot; + values[0]);</td>
</tr>
<tr>
<td>log</td>
<td>A reference to the logging subsystem. Logging can be configured and log messages are stored in files, which makes it a bit more flexible than simply using</td>
</tr>
</tbody>
</table>
Transformations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;out&quot;</td>
<td>Here's how you write a few log messages with varying severities:</td>
</tr>
<tr>
<td></td>
<td>log.debug(&quot;This is a DEBUG message, it will probably be disregarded&quot;);</td>
</tr>
<tr>
<td></td>
<td>log.info(&quot;This is a INFO message, it will probably be written to the logs&quot;);</td>
</tr>
<tr>
<td></td>
<td>log.warn(&quot;This is a WARN message, it will most likely be written to the logs&quot;);</td>
</tr>
<tr>
<td></td>
<td>log.error(&quot;This is a ERROR message, it will almost certainly be written to the logs&quot;);</td>
</tr>
</tbody>
</table>

Data types:

### Table 5.2. JavaScript data types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>String values are represented as JavaScript strings, which means that they have (among others) methods like:</td>
</tr>
<tr>
<td></td>
<td>var str = values[0];</td>
</tr>
<tr>
<td></td>
<td>// get the length of a string var len = str.length();</td>
</tr>
<tr>
<td></td>
<td>// uppercase variant of a string var up = str.toUpperCase();</td>
</tr>
<tr>
<td></td>
<td>// lowercase variant of a string var lw = str.toLowerCase();</td>
</tr>
<tr>
<td></td>
<td>For more information, we recommend W3 schools JavaScript string reference [<a href="http://www.w3schools.com/jsref/jsref_obj_string.asp">http://www.w3schools.com/jsref/jsref_obj_string.asp</a>].</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Numbers are treated as regular JavaScript numbers, which means that they have (among others) methods and operators like:</td>
</tr>
<tr>
<td></td>
<td>var num = values[0];</td>
</tr>
<tr>
<td></td>
<td>// format with 2 decimals var formattedNumber = num.toFixed(2);</td>
</tr>
<tr>
<td></td>
<td>// add, subtract, multiply or divide var m = (4 + num * 2 - 1) / 2;</td>
</tr>
</tbody>
</table>
|           | For more information, we recommend W3 schools JavaScript number reference [http://www.w3schools.com/jsref/jsref_obj_number.asp] and also check out the Math function [http://
## Transformations

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="http://www.w3schools.com/jsref/jsref_obj_math.asp">www.w3schools.com/jsref/jsref_obj_math.asp</a> reference.</td>
</tr>
</tbody>
</table>
| DATE      | Date values are treated as Java dates, which is a bit unusual, but leaves you with almost an identical interface as a regular JavaScript date. Here’s a summary of typical methods:  

```javascript
var d = values[0];
var year = d.getYear();
var month = d.getMonth();
var date = d.getDate();
var hour = d.getHour();
var minutes = d.getMinutes();
var seconds = d.getSeconds();

// milliseconds since 1970-01-01
var timestamp = d.getTime();
```

For a full reference, please look at the Java Date class reference [http://download.oracle.com/javase/6/docs/api/java/util/Date.html]. |
| BOOLEAN   | Boolean (true/false) values are simply booleans, no sugar coating added :) |

### EasyDQ services

EasyDataQuality [http://www.easydq.com] is an on-demand service for data quality functions. DataCleaner provides access to the EasyDQ services, but the core functionality is provided by Human Inference.

Please refer to the EasyDQ for DataCleaner documentation [http://help.easydq.com/datacleaner] for detailed information about the services provided through EasyDQ.
Equals

The *Equals* filter provides a way to make a simple filtering condition based on a white list / valid list of values. Simply enter a list of values that you accept for a given column, and then you can map your flow to the VALID outcome of the filter.

Here’s an example of an Equals filter configuration where valid Gender values are being checked.
Use the plus/minus buttons to grow or shrink the list of values you want to accept.

If placed as the first component in a flow, the Equals filter is optimizable in a way where it will modify your originating query. This means that it is also an appropriate filter to use if you just want to sample the data used in your job.

**Max rows**

The Max rows filter is used to limit the amount of records that are passed further on in the job’s flow.

If placed as the first component in a flow, the Max rows filter is optimizable in a way where it will modify your originating query. This means that it is also an appropriate filter to use if you just want to sample the data used in your job.

**Not null**

The Not null filter is a simple filter that can be used to exclude null values from your flow. Additionally you can select whether or not you want to accept empty strings ("") or not.

If placed as the first component in a flow, the Not null filter is optimizable in a way where it will modify your originating query. This means that it is also an appropriate filter to use if you just want to sample the data used in your job.
Chapter 6. Writers

Abstract

Although the primary focus of DataCleaner is analysis, often during such analysis you will find yourself actually improving data by means of applying transformers and filters on it. When this is the case, obviously you will want to export the improved/cleansed data so you can utilize it in other situations than the analysis.

Writing cleansed data is achieved by wiring the job together with an output writer instead of (or in addition to) an analyzer. Adding an output writer is done by selecting it from the "Write data" menu item. Alternatively by clicking either your filter's outcome buttons or your transformer's "Save transformed data" button. When writing data there's a couple of available output formats:

In the following sections each output format option will be described:

Create CSV file

 Writes a data set to an Comma Separated Values file. CSV files are a popular choise for interoperability with other systems and loading of data into databases.

Create Excel spreadsheet

 Writes a data set to an Excel spreadsheet. An advantage of this approach is that a single file can contain multiple sheets, and that it is easily navigatable in Microsoft Excel. A disadvantage is that for very large data sets it is less performant.

Create staging table

 Writes a data set to an embedded relational database, which DataCleaner manages. This option is primarily used for staging data for further analysis. The advantage of using the feature is that it retains column type information, it can handle a lot of data and multiple data sets can be written to the same datastore. A disadvantage is that the data is not easily readable by third party applications (unless exported again).
Insert into table

Using this writer you can insert your data into a table of an existing datastore. If you already have a table layout ready or if you want to append to eg. a database table, then this writing option is the right one for you.

Optionally, you can make the 'Insert into table' component truncate your table before insertion. This will delete all existing records in the table, useful for initial load situations.

Currently target tables can be from any of the following datastore types:

1. *CSV file*. In this case data will be appended to the file.
2. *Excel spreadsheet*. In this case data will be appended to the file.
3. *Relational database*. In this case data will be inserted to the table using an INSERT statement.
4. *MongoDB database*. In this case data will be inserted into the MongoDB collection.
5. *CouchDB database*. In this case data will be inserted into the CouchDB database.
Part III. Reference data
Table of Contents

7. Dictionaries .................................................................................................................. 40
8. Synonyms (aka. Synonym catalogs) ............................................................................. 41
   Text file synonym catalog ......................................................................................... 41
   Datastore synonym catalog ...................................................................................... 41
9. String patterns ............................................................................................................ 42
Chapter 7. Dictionaries

Dictionaries are reference data lists used for verifying or categorizing values against certain black- or whitelists. Dictionaries are generally enumerable and finite, whereas eg. string patterns are dynamic and evaluated each time.

Examples of meaningful dictionaries are:

1. A dictionary of product types like "jewelry", "menswear", "sportswear" etc.
2. A dictionary of gender symbols like "M", "F" and maybe "UNKNOWN".
3. A dictionary of age group names (eg. infant, child, young, mature, senior)
4. Two dictionaries for male and female given names (in order to determine gender of persons)
Chapter 8. Synonyms (aka. Synonym catalogs)

Abstract

Synonym catalogs are used to replace and standardize values to their master terms, in order to avoid multiple terms for the same real world thing.

There are many real life examples of synonyms that make for messy data, for example:

1. Company and brand names, like "Coca-Cola", "Coca cola" and "Coke".
2. Titles, like "Doctor", "Dr." and "Doc"

In the following sections we will describe how to set up synonym catalogs that can be used in a variety of ways to standardize your database.

Text file synonym catalog

A text file synonym catalog is the easiest and often also the fastest way to perform synonym replacement. Simply create a text file with content in a format, where the master term is succeeded with a comma-separated list of synonyms, like this:

M,Male,Man,Guy,Boy
F,Female,Woman,Girl

In the above example, most typical gender tokens will be replaced with either "M" or "F".

Datastore synonym catalog

If your synonyms are located in a database or another type of datastore, then you can also create synonym catalogs based on this.

Datastore synonym catalogs allow you to specify a single master term column and multiple synonym columns. The synonym catalog will look then find synonym matches by searching/querying the datastore.
Chapter 9. String patterns

String patterns define a "template" for string values which they may or may not conform to.

DataCleaner currently supports two type of popular string formats:

1. Regular expressions, which is a general purpose string pattern matching language popular in computer science. Regular expressions does take a bit of time to learn, but are very powerful once harnessed.

   Explaining the syntax of regular expressions is definitely outside the scope of the DataCleaner documentation. We recommend the Java Regular Expressions lesson [http://docs.oracle.com/javase/tutorial/essential/regex/] if you are looking for a resource on this.

2. Simple string patterns, which use the same syntax as the Pattern finder analyzer. Patterns such as "aaaa@aaaa.aaa" could for example be used to match typical email addresses.
Part IV. Configuration reference
# Table of Contents

10. Configuration file ................................................................. 45
    XML schema ........................................................................... 45
    Datastores .............................................................................. 45
    Database (JDBC) connections ..................................................... 45
    Comma-Separated Values (CSV) files ............................................ 46
    Fixed width value files ............................................................. 46
    Excel spreadsheets ................................................................... 47
    XML file datastores ................................................................. 47
    MongoDB databases ................................................................. 48
    CouchDB databases ................................................................... 49
    Reference data .......................................................................... 50
    Task runner .............................................................................. 50
    Storage provider ....................................................................... 50

11. Analysis job files ..................................................................... 52
    XML schema ........................................................................... 52
    Source section .......................................................................... 52

12. Logging .................................................................................... 54
    Logging configuration file ........................................................... 54
    Default logging configuration ....................................................... 54
    Modifying logging levels .............................................................. 55
    Alternative logging outputs .......................................................... 56
Chapter 10. Configuration file

Abstract

In this chapter we go through the elements of a configuration file, conf.xml, making it possible (although optional) to change the static configuration and configure the environment of DataCleaner.

In the DataCleaner monitoring web application, the conf.xml file is the only point of configuration. The file is located in the root of each tenant's repository folder. For more information, refer to the repository chapter.

Most of the elements in the configuration file is also editable within the Desktop application. It is however important to note that changes made in the GUI are not saved directly to the configuration file, but to the userpreferences.dat file. You can consider the relationship between the two files this way: The configuration file defines a static, unmodifyable prototype of the applications environment. All customizations made to this prototype in the Desktop application is saved in the userpreferences.dat file.

XML schema

The configuration file (conf.xml) is an XML file pertaining to the XML namespace "http://eobjects.org/ analyzerbeans/configuration/1.0".


Datastores

Datastores can be configured in the configuration file under the element <datastore-catalog>. The following sections will go into further details with particular types of datastores.

Database (JDBC) connections

Here are a few examples of common database types.

Tip

The DataCleaner User Interface makes it a lot easier to figure out the url (connection string) and driver class part of the connection properties. It's a good place to start if you don't know these properties already.

MySQL

<jdbc-datastore name="MySQL datastore">
    <driver>com.mysql.jdbc.Driver</driver>
    <username>username</username>
    <password>password</password>
</jdbc-datastore>

Oracle
<jdbc-datastore name="Oracle datastore">
  <url>jdbc:oracle:thin:@hostname:1521:sid</url>
  <driver>oracle.jdbc.OracleDriver</driver>
  <username>username</username>
  <password>password</password>
</jdbc-datastore>

Microsoft SQL Server

A typical connection to Microsoft SQL server will look like this:

<jdbc-datastore name="MS SQL Server datastore">
  <url>jdbc:jtds:sqlserver://hostname/database;useUnicode=true;characterEncoding=UTF-8</url>
  <driver>net.sourceforge.jtds.jdbc.Driver</driver>
  <username>username</username>
  <password>password</password>
</jdbc-datastore>

However, if you want to use an instance name based connection, then the SQL Server Browser service MUST BE RUNNING and then you can include the instance parameter: Here's an example for connecting to a SQLEXPRESS instance:

<url>jdbc:jtds:sqlserver://hostname/database;instance=SQLEXPRESS;useUnicode=true;characterEncoding=UTF-8</url>

Comma-Separated Values (CSV) files

This is an example of a CSV file datastore

<csv-datastore name="my_csv_file">
  <filename>/path/to/file.csv</filename>
  <quote-char>"</quote-char>
  <separator-char>;</separator-char>
  <encoding>UTF-8</encoding>
  <fail-on-inconsistencies>true</fail-on-inconsistencies>
  <header-line-number>1</header-line-number>
</csv-datastore>

Fixed width value files

Files with fixed width values can be registered in two ways - either with a single fixed-width size for all columns, or with individual value-widths.

Here's an example with a fixed width specification for all columns:

<fixed-width-datastore name="My file">
  <filename>foobar.txt</filename>
  <width-specification>
    <fixed-value-width>20</fixed-value-width>
  </width-specification>
<encoding>UTF-8</encoding>
<fail-on-inconsistencies>true</fail-on-inconsistencies>
</fixed-width-datastore>

Here's an example with individual (3 columns) width specifications:

<fixed-width-datastore name="My file">
  <filename>foobar.txt</filename>
  <width-specification>
    <value-width>4</value-width>
    <value-width>17</value-width>
    <value-width>19</value-width>
  </width-specification>
  <encoding>UTF-8</encoding>
  <fail-on-inconsistencies>true</fail-on-inconsistencies>
</fixed-width-datastore>

Excel spreadsheets

This is an example of a Excel spreadsheet datastore

<excel-datastore name="my_excel_spreadsheet">
  <filename>/path/to/file.xls</filename>
</excel-datastore>

XML file datastores

Defining XML datastores can be done in both a simple (automatically mapped) way, or an advanced (and more performant and memory effective way).

The simple way is just to define a xml-datastore with a filename, like this:

<xml-datastore name="my_xml_datastore">
  <filename>/path/to/file.xml</filename>
</xml-datastore>

This kind of XML datastore works find when the file size is small and the hierarchy is not too complex. The downside to it is that it tries to automatically detect a table structure that is fitting to represent the XML contents (which is a tree structure, not really a table).

To get around this problem you can also define your own table structure in which you specify the XPaths that make up your rows and the values within your rows. Here's an example:

<xml-datastore name="my_xml_datastore">
  <filename>/path/to/file.xml</filename>
  <table-def>
    <rowXpath>/greetings/greeting</rowXpath>
    <valueXpath>/greetings/greeting/how</valueXpath>
    <valueXpath>/greetings/greeting/what</valueXpath>
  </table-def>
</xml-datastore>
The datastore defines a single table, where each record is defined as the element which matches the XPath "/greetings/greeting". The table has two columns, which are represented by the "how" and "what" elements that are child elements to the row's path.

For more details on the XPath expressions that define the table model of XML datastores, please refer to MetaModel's tutorial on the topic [http://metamodel.eobjects.org/example_xml_mapping.html] (MetaModel is the data access library used to read data in DataCleaner).

**MongoDB databases**

This is an example of a fully specified MongoDB datastore, with an example table structure based on two collections.

```xml
<mongodb-datastore name="my_mongodb_datastore">
  <hostname>localhost</hostname>
  <port>27017</port>
  <database-name>my_database</database-name>
  <username>user</username>
  <password>pass</password>
  <table-def>
    <collection>company_collection</collection>
    <property>
      <name>company_name</name>
      <type>VARCHAR</type>
    </property>
    <property>
      <name>customer</name>
      <type>BOOLEAN</type>
    </property>
    <property>
      <name>num_employees</name>
      <type>INTEGER</type>
    </property>
    <property>
      <name>address_details</name>
      <type>MAP</type>
    </property>
  </table-def>
  <table-def>
    <collection>person_collection</collection>
    <property>
      <name>person_name</name>
      <type>VARCHAR</type>
    </property>
    <property>
      <name>birthdate</name>
      <type>DATE</type>
    </property>
    <property>
      <name>emails</name>
      <type>LIST</type>
    </property>
  </table-def>
</mongodb-datastore>
```
If the hostname and port elements are left out, localhost:27017 will be assumed.

If the username and password elements are left out, an anonymous connection will be made.

If there are no table-def elements, the database will be inspected and table definitions will be auto-detected based on the first 1000 documents of each collection.

**CouchDB databases**

This is an example of a fully specified CouchDB datastore, with an example table structure based on two CouchDB databases.

```xml
<couchdb-datastore name="my_couchdb_datastore">
  <hostname>localhost</hostname>
  <port>5984</port>
  <username>user</username>
  <password>pass</password>
  <ssl>true</ssl>
  <table-def>
    <database>company_collection</database>
    <field>
      <name>company_name</name>
      <type>VARCHAR</type>
    </field>
    <field>
      <name>customer</name>
      <type>BOOLEAN</type>
    </field>
    <field>
      <name>num_employees</name>
      <type>INTEGER</type>
    </field>
    <field>
      <name>address_details</name>
      <type>MAP</type>
    </field>
  </table-def>
  <table-def>
    <database>person_collection</database>
    <field>
      <name>person_name</name>
      <type>VARCHAR</type>
    </field>
    <field>
      <name>birthdate</name>
      <type>DATE</type>
    </field>
    <field>
      <name>emails</name>
    </field>
  </table-def>
</couchdb-datastore>
```
Configuration file

If the hostname and port elements are left out, localhost:5984 will be assumed.

If the username and password elements are left out, an anonymous connection will be made.

If the "ssl" element is false or left out, a regular HTTP connection will be used.

If there are no table-def elements, the database will be inspected and table definitions will be auto-detected based on the first 1000 documents of each database.

Reference data

Reference data are defined in the configuration file in the element <reference-data-catalog>.

Task runner

The task runner defines how DataCleaner's will engine will execute the tasks of an analysis job. Typically you shouldn't edit this element. However, here are the two options:

<multithreaded-taskrunner max-threads="30" />

Defines a multi threaded task runner with a thread pool of 30 available threads. Beware that although 30 might seem like a high number that too small a pool of threads might cause issues because some tasks schedule additional tasks and thus there's a risk of dead lock when thread count is very low.

<singletaskrunner />

Defines a single threaded task runner. On legacy hardware or operating systems this setting will be better, but it will not take advantage of the multi threading capabilities of modern architecture.

Storage provider

The storage provider is used for storing temporary data used while executing an analysis job. There are two types of storage: Large collections of (single) values and "annotated rows", ie. rows that have been marked with a specific category which will be of interest to the user.

To explain the storage provider configuration let's look at the default element:

<storage-provider>
  <combined>
    <collections-storage>
      <berkeley-db/>
    </collections-storage>
    <row-annotation-storage>
      <in-memory max-rows-threshold="1000"/>
    </row-annotation-storage>
  </combined>
</storage-provider>
The element defines a combined storage strategy.

Collections are stored using berkeley-db, an embedded database by Oracle. This is the recommended strategy for collections.

Row annotations are stored in memory. There's a threshold of 1000 rows which means that if more than 1000 records are annotated with the same category then additional records will not be saved (and thus is not viewable by the user). Most user scenarios will not require more than max. 1000 annotated records for inspection, but if this is really necessary a different strategy can be pursued:

**Using MongoDB for annotated rows**

If you have a local MongoDB [http://www.mongodb.org/] instance, you can use this as a store for annotated rows. This is how the configuration looks like:

```xml
<row-annotation-storage>
  <custom-storage-provider class-name="org.eobjects.analyzer.storage.MongoDbStorageProvider"/>
</row-annotation-storage>
```

The MongoDB storage provider solution has shown very good performance metrics, but does add more complexity to the installation, which is why it is still considered experimental and only for savvy users.
Chapter 11. Analysis job files

Abstract

Job files contain the information about the execution of a DataCleaner job. Typically these files have the file extension `.analysis.xml`. In this file we will explain the file format, which is XML based, and explain how it relates to what DataCleaner does.

A job will always reference items in a configuration, such as datastores, reference data and more. Therefore a job alone is not enough to execute. But multiple jobs can use the same configuration. For more information on the configuration, see the configuration file chapter.

XML schema

Analysis job files are written in an XML format pertaining to the XML namespace "http://eobjects.org/analyzerbeans/job/1.0".

For XML-savvy readers, who prefer to use XML schema aware editors to edit their XML files, you can find the XML schema for this namespace here: http://eobjects.org/svn/AnalyzerBeans/trunk/env/xml-config/src/main/resources/job.xsd.

Read on in this chapter for notes on individual parts of the job file format.

Source section

The source section of the job file format is probably the most interesting one to manually edit or review. Here's an example source section:

```xml
<source>
  <data-context ref="orderdb" />
  <columns>
    <column path="PUBLIC.EMPLOYEES.EMPLOYEENUMBER" id="col_0" type="INTEGER" />
    <column path="PUBLIC.EMPLOYEES.LASTNAME" id="col_1" type="VARCHAR" />
    <column path="PUBLIC.EMPLOYEES.FIRSTNAME" id="col_2" type="VARCHAR" />
    <column path="PUBLIC.EMPLOYEES.EXTENSION" id="col_3" type="VARCHAR" />
    <column path="PUBLIC.EMPLOYEES.EMAIL" id="col_4" type="VARCHAR" />
    <column path="PUBLIC.EMPLOYEES.OFFICECODE" id="col_5" type="VARCHAR" />
    <column path="PUBLIC.EMPLOYEES.REPORTSTO" id="col_6" type="INTEGER" />
    <column path="PUBLIC.EMPLOYEES.JOBTITLE" id="col_7" type="VARCHAR" />
  </columns>
  <variables>
    <variable id="employee type" value="Sales Rep" />
  </variables>
</source>
```

From this source section we can derive these interesting facts:

1. The job is using the datastore called 'orderdb'. How this datastore is configured, the job is not aware of, since it is defined in the configuration. Potentially the job could be used with multiple similar datastores, as long as their name was 'orderdb'.
2. The columns defined make up the base of the source query that the job will fire. Each column is assigned an artificial ID, and a hint about it's data type is provided. This information is there to be able to detach or replace a column with a new definition. That means that if you've spend a long time building the perfect job, but want to apply it to a different column, you can potentially "just" change the column definition here and retain the original column ID.

3. In this source section we also see some variables. This is an optional sub-section and not that common. The variables are property values that can be replaced at runtime with new values. See the chapter Parameterizable jobs for more information and examples.
Chapter 12. Logging

Abstract

Logging in DataCleaner is configurable either by supplying an XML file or a properties file. In this chapter we explore logging configuration and how you can fine-tune logging to your needs.

Logging configuration file

Logging in DataCleaner is based on Log4j, an open source logging framework by the Apache foundation. With log4j you can configure logging at a very detailed level, while at the same time keeping a centralized configuration.

There are three approaches to configuring logging in DataCleaner:

1. The default logging configuration. This requires no changes to the standard distribution of DataCleaner. Log files will be generated in the log/datacleaner.log file.

2. Specifying your own XML log configuration. This requires you to put a file named log4j.xml in the root directory of DataCleaner.

3. Specifying your own property file log configuration. This requires you to put a file named log4j.properties in the root directory of DataCleaner.

The recommended way of doing custom configuration of DataCleaner logging is using the XML format. In the following sections we will explain this approach using examples. For more detailed documentation on Log4j configuration, please refer to the Log4j website [http://logging.apache.org/log4j/].

Default logging configuration

Here's a listing of the default logging configuration, in XML format:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
  <appender name="consoleAppender" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%5p %d{HH:mm:ss} %c{1} - %m%n"/>
    </layout>
  </appender>

  <appender name="fileAppender" class="org.apache.log4j.DailyRollingFileAppender">
    <param name="File" value="log/datacleaner.log"/>
    <param name="DatePattern" value=".'yyyy-MM-dd'.log'"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%5p %d{HH:mm:ss.SSS} %c{1} - %m%n"/>
    </layout>
  </appender>
</log4j:configuration>
```
<logger name="org.eobjects.metamodel">  
  <level value="info" />  
</logger>  

<logger name="org.eobjects.analyzer">  
  <level value="info" />  
</logger>  

<logger name="org.eobjects.analyzer.job.runner">  
  <level value="info" />  
</logger>  

<logger name="org.eobjects.analyzer.storage">  
  <level value="info" />  
</logger>  

<logger name="org.eobjects.analyzer.descriptors.ClasspathScanDescriptorProvider">  
  <level value="info" />  
</logger>  

<logger name="org.eobjects.datacleaner">  
  <level value="info" />  
</logger>  

<root>  
  <priority value="info" />  
  <appender-ref ref="consoleAppender" />  
  <appender-ref ref="fileAppender" />  
</root>  

</log4j:configuration>  

This logging configuration specifies the INFO level as the default level of logging. It appends (outputs) log messages to the console (if available) and to a file with the path: log/datacleaner.log  

We recommend using this default configuration as a template for custom log configurations. Next we will explore how to modify the configuration and create new logging outputs.  

**Modifying logging levels**  

These are the logging levels available in DataCleaner and Log4j, order by priority (highest priority first):  

1. error  
2. warn  
3. info  
4. debug  
5. trace  

Typically the bottom-two logging levels (debug and trace) are not used unless unexpected situations has to be investigated by developers.
Modifying the logging levels can be done either globally or in a hierarchical manner:

1. If you change the <priority> element's value attribute, you change the global threshold for logging messages.

2. If you change the <logger> element's level, you change the logging priority logging messages that pertain to a particular hierarchy of loggers.

### Alternative logging outputs

Log messages are printed to an output, typically a file or the console. In the configuration file this is configured in the <appender> elements. Here's a few examples of alternative appenders you can use. For more examples and documentation, please refer to the Log4j website [http://logging.apache.org/log4j/].

**Logging in a PostgreSQL database:**

```xml
<appender name="jdbcAppender" class="org.apache.log4j.jdbc.JDBCAppender">
  <param name="URL" value="jdbc:postgresql:db"/>
  <param name="Driver" value="org.postgresql.Driver"/>
  <param name="User" value="user"/>
  <param name="Password" value="password"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="INSERT INTO log4j (log_date,log_level,log_location,log_message) VALUES ('%d{yyyy-MM-dd}', '%p', '%C;%L', '%m')"/>
  </layout>
</appender>
```
Part V. DataCleaner monitor repository
# Table of Contents

13. Repository configuration ................................................................. 59  
   Configure repository location ......................................................... 59  
   Providing signed Java WebStart client files ...................................... 59  
   Cluster configuration (distributed execution) ..................................... 60  
14. Repository layout .......................................................................... 62  
   Multi-tenant layout .......................................................................... 62  
   Tenant home layout .......................................................................... 62
Chapter 13. Repository configuration

Abstract

In this chapter we will explain configuration of the repository of the DataCleaner monitor web application. By default the repository and other artifacts are bundled with the application, but for production deployments this configuration may not be sufficient. Learn about how to deploy a repository that is located independently of the web application code.

Configure repository location

By default DataCleaner monitor web application uses a file-based repository location which is relative to the deployed web archive. This makes it easy to deploy and test-drive, but it might not be the best production deployment choice.

To change the repository location, find the file WEB-INF/classes/context/repository-context.xml within the deployed web archive folder. Find the <bean> element which initially looks like this:

```xml
<bean name="repository" class="org.eobjects.datacleaner.repository.file.FileRepository">
  <constructor-arg type="java.io.File" value="repository"/>
</bean>
```

You can modify the value attribute of the inner <constructor-arg> element to point to a directory location of choice. The location can be absolute or relative to the web archive folder. For instance you might want to use a configuration like this in a unix environment:

```xml
<bean name="repository" class="org.eobjects.datacleaner.repository.file.FileRepository">
  <constructor-arg type="java.io.File" value="/var/datacleaner/repository"/>
</bean>
```

Tip

The enterprise edition of DataCleaner also includes a database backed repository - this makes larger deployments both faster, easier to administer and scalable to multiple machines.

Providing signed Java WebStart client files

The DataCleaner monitor web application features an option to let the user launch the desktop application for editing and testing jobs deployed on the monitor server. To enable this special mode of interoperability, signed JAR files needs to be provided, since otherwise the desktop application will not be allowed to launch by most Java runtime configurations.

Signed JAR files can be downloaded separately [http://datacleaner.org/downloads] and should be extracted into a directory of choice on the server. Once extracted you need to configure the application with the directory path. Find the file WEB-INF/classes/context/repository-context.xml within the deployed web archive folder. Remove or comment the <bean> element which initially looks like this:

```xml
<bean name="launchArtifactProvider" lazy-init="true"
      class="org.eobjects.datacleaner.monitor.server.DevModeLaunchArtifactProvider"/>
```

Repository configuration

It needs to be replaced with a new `<bean>` element which will look like this:

```xml
<bean name="launchArtifactProvider"
      lazy-init="true" class="org.eobjects.datacleaner.monitor.server.FileFolderLaunchArtifactProvider">
  <constructor-arg type="java.io.File" value="/var/datacleaner/signed_jars"/>
</bean>
```

Cluster configuration (distributed execution)

DataCleaner monitor allows execution of jobs through a cluster of machines - essentially to increase fault tolerance and performance by adding more machines instead of having to upgrade the hardware of a single machine.

When executing distributed jobs, DataCleaner will initially estimate how many records needs to be processed. Depending on this number, a number of "chunks" of records will be assigned to be executed on different slave execution nodes. After execution, the master node will collect the slave node results and combine them into a single result report.

The configuration of DataCleaner's cluster is handled through the file `WEB-INF/classes/context/cluster-context.xml` within the deployed web archive folder. By default it defines this `<bean>` element:

```xml
<bean id="clusterManagerFactory" class="org.eobjects.datacleaner.monitor.cluster.HttpClusterManagerFactory">
  <property name="username" value="admin" />
  <property name="password" value="admin" />
  <property name="slaveServerUrls">
    <list>
      <value>http://localhost:8080/DataCleaner-monitor</value>
      <value>http://localhost:9090/DataCleaner-monitor</value>
    </list>
  </property>
</bean>
```

The above definition states that the cluster has two slave execution nodes. As an example, these are using 'localhost' references, but you can also use other hostnames.

To enable clustered execution of a job, you need to open it's `.schedule.xml` file in the 'jobs' folder of the repository. In this XML file you will find a `<distributed-execution>` element which determines if local or distributed execution will be executed. For example, the file 'Customer completeness.schedule.xml' starts like this:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<schedule
 xmlns="http://eobjects.org/datacleaner/schedule/1.0"
 xmlns:ns2="http://eobjects.org/datacleaner/shared/1.0"
 xmlns:ns3="http://eobjects.org/datacleaner/timeline/1.0"
 xmlns:ns4="http://eobjects.org/datacleaner/execution-log/1.0">
  <cron-expression>@daily</cron-expression>
  <distributed-execution>false</distributed-execution>
  <alerts>
    ...
```
Changing this value to 'true' would trigger DataCleaner monitor to use the cluster configuration when executing the job.

**Tip**

The *enterprise edition* of DataCleaner also include other mechanisms of communication between cluster nodes. One short-coming of the above approach is that it is not tolerant to network issues or crashing nodes. Consider DataCleaner enterprise edition for such deployments, since it supports elastic clusters without having the master to be aware of each single node.
Chapter 14. Repository layout

Abstract

In this chapter we look at the file and folder layout of a DataCleaner monitor repository. Beginning with the multi-tenant layout, and then proceeding with a typical tenant's repository layout.

Multi-tenant layout

The DataCleaner repository layout, used by the monitoring web application, is built to support multi-tenant deployments. Therefore, on the root level of the repository, folders are located which each represent a tenant's separate home folder. Users from one tenant are not able to access files or folders from other tenant's home folders.

Tenant home layout

To function properly, each tenant home folder requires these files and folders:

1. conf.xml (file)
2. jobs (folder)
3. results (folder)
4. timelines (folder)

The conf.xml file represents the DataCleaner configuration for the particular tenant. The file format is the same as described in the Configuration file chapter. It is recommended to use the supplied example conf.xml file (for the 'DC' tenant) as a template for further customization. Specifically the custom elements for task-runner, descriptor-provider and storage-provider in this template conf.xml file is recommended for optimal performance.

The folders are all managed by the DataCleaner monitoring web application, so only in rare cases should you manually interact with them.

It is allowed to add more files and folders to the tenant home. These will not be managed by the monitor application, but can be referenced eg. as the filename paths of datastores defined in conf.xml.
Part VI. DataCleaner
monitor web services
# Table of Contents

15. Job triggering ........................................................................................................ 65  
   Trigger service ........................................................................................................ 65  
   Polling for execution status ................................................................................... 66  
16. Repository navigation ............................................................................................ 67  
   Job files .................................................................................................................... 67  
   Result files ................................................................................................................. 67  
   Uploading content to the repository ....................................................................... 68  
   Modifying result metadata ...................................................................................... 68  
   Renaming jobs ......................................................................................................... 69  
   Copying jobs ............................................................................................................ 69  
   Deleting jobs ........................................................................................................... 70  
17. Metric web services ............................................................................................... 71  
   Metrics background ............................................................................................... 71  
   Getting a list of available metrics .......................................................................... 71  
   Getting the values of particular metrics ................................................................ 71  
18. Atomic transformations (data cleaning as a service) ............................................. 73  
   What are atomic transformation services? .............................................................. 73  
   Invoking atomic transformations ........................................................................... 73
Chapter 15. Job triggering

Abstract

The DataCleaner web monitor usually handles scheduling of jobs, but sometimes you need to trigger them manually. And sometimes third party systems need to trigger them manually.

In this chapter, a web service for triggering DataCleaner jobs in the DataCleaner monitor web application is explained.

Trigger service

The web service is reachable by this HTTP URL:

/DataCleaner-monitor/repository/{tenant}/jobs/{job}.trigger

The response from the trigger web service is a JSON document that looks like this:

```
{
    "status":"PENDING",
    "logOutput":null,
    "endDate":null,
    "beginDate":null,
    "resultId":"Customer completeness-1349876418933",
    "triggeredBy":"admin"
}
```

Optionally you can also add these parameters:

**Table 15.1. Job triggering HTTP parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>Set to ‘true’ if the server should wait until the job has executed before responding. This will ensure that the client is able to read a ‘status’ which is either ‘SUCCESS’ or ‘FAILURE’ (unless a timeout occurs).</td>
</tr>
<tr>
<td>timeoutMillis</td>
<td>To be used in combination with ‘block’. Set this to the max number of milliseconds to wait before responding.</td>
</tr>
</tbody>
</table>

Using these parameters, a typical response will look like this:

```
{
    "status":"SUCCESS",
    "logOutput":
    "2012-10-10 16:04:02 - Job execution BEGIN
```
Polling for execution status

Often times it is not practical to use the 'block' service parameter, if the job is a long-running job. Rather then you can retrieve the status of the job execution using this service URL:

/DataCleaner-monitor/repository/{tenant}/logs/{resultId}

The 'resultId' part in this URL is mean to match the 'resultId' coming from the responses of the trigger service.

The response of the service will be an XML representation of the execution status. For instance:

```xml
  <ns4:result-id>Customer duplicates-1347004507995</ns4:result-id>
  <ns4:job-begin-date>2012-09-07T09:55:10.607+02:00</ns4:job-begin-date>
  <ns4:job-end-date>2012-09-07T09:55:19.661+02:00</ns4:job-end-date>
  <ns4:execution-status>SUCCESS</ns4:execution-status>
  <ns4:trigger-type>MANUAL</ns4:trigger-type>
  <ns4:triggered-by>admin</ns4:triggered-by>
  <schedule>
  </schedule>
</ns4:execution-log>
```
Chapter 16. Repository navigation

Abstract

In this chapter we will learn how to use web service URLs to navigate the DataCleaner repository. The repository is available at the root level of the DataCleaner monitoring web application. Access is of course restricted to the tenant's home folder. Therefore, all these web services are located with URLs starting with the form:

/DataCleaner-monitor/repository/DC/...

Where 'DC' is the tenant identifier and 'DataCleaner-monitor' is the web application archive name.

Job files

Job files are available in the reserved folder name 'jobs'. To get a listing of all job files (in JSON format), go to:

/DataCleaner-monitor/repository/DC/jobs

The result will resemble this example result:

```
[
  {
    "repository_path": "/DC/jobs/Contributor name cleansing.analysis.xml",
    "name": "Contributor name cleansing","filename": "Contributor name cleansing.xml",
  },
  {
    "repository_path": "/DC/jobs/Customer completeness.analysis.xml",
    "name": "Customer completeness","filename": "Customer completeness.analysis.xml",
  },
  {
    "repository_path": "/DC/jobs/Customer duplicates.analysis.xml",
    "name": "Customer duplicates","filename": "Customer duplicates.analysis.xml",
  }
]
```

Further navigating to one of these repository paths, you will be able to read the full XML description of the job, as described in the Analysis Job files chapter:

/DataCleaner-monitor/repository/DC/jobs/product_profiling.analysis.xml

Where 'DC' is the tenant identifier, 'product_profiling' is the name of the job and 'DataCleaner-monitor' is the web application archive name.

Result files

Result files are available in the reserved folder name 'results'. To get a listing of all job files (in JSON format), go to:

/DataCleaner-monitor/repository/DC/results
Repository navigation

/DataCleaner-monitor/repository/DC/results

Where 'DC' is the tenant identifier and 'DataCleaner-monitor' is the web application archive name.
The result will resemble this example result:

```
[  
    {"repository_path": "/DC/results/Customer completeness-1345128427583.analysis.result.dat", 
     "filename": "Customer completeness-1345128427583.analysis.result.dat"},  
    {"repository_path": "/DC/results/Customer completeness-1345200106074.analysis.result.dat", 
     "filename": "Customer completeness-1345200106074.analysis.result.dat"}  
]
```

Further navigating to one of these repository paths, you will be able to inspect the analysis result, rendered in HTML format.

/DataCleaner-monitor/repository/DC/results/Customer completeness-1345200106074.

Where 'DC' is the tenant identifier, 'Customer completeness' is the name of the job, '1345200106074' is the timestamp of the result and 'DataCleaner-monitor' is the web application archive name.

Uploading content to the repository

Files can be uploaded to the repository using HTTP POST upload requests. The types of uploadable files are:

1. DataCleaner jobs (.analysis.xml files), which are uploaded using this URL template:
   
   POST /DataCleaner-monitor/repository/{tenant}/jobs/{job}.analysis.xml

   Where the request parameter 'file' is a multi-part file containing the new job file content.

2. DataCleaner results (.analysis.result.dat files), which are uploaded using this URL template:

   POST /DataCleaner-monitor/repository/{tenant}/results/{result}.analysis.result.dat

   Where the request parameter 'file' is a multi-part file containing the new result file content.

3. Configuration (conf.xml file), which are uploaded using this URL template:

   POST /DataCleaner-monitor/repository/{tenant}/conf.xml

   Where the request parameter 'file' is a multi-part file containing the configuration file content.

Modifying result metadata

In some cases you can end up having results in the DataCleaner monitor repository which are not correctly configured. Specifically we sometimes see these situations:
1. The registered date of the result might not be correct, if the result was created at a later time than the data it represents.

2. The name of the associated job is wrong, if the result was uploaded from the desktop application but with a reference to a client-side job.

To fix this situation, there is a result modification web service available. The service requires the ADMIN security role.

The request URL is the same as the URL for inspecting the result, but where `.analysis.result.dat` is replaced with `.modify`:

```
POST /DataCleaner-monitor/repository/{tenant}/results/{result}.modify
```

The POST request body needs to be a JSON document, specifying the new metadata for the result file. Here's an example:

```
{
    "job":"name of new job",
    "date":"2012-10-17 21:47:00",
    "overwrite":true
}
```

### Renaming jobs

Renaming a job in the DataCleaner monitor repository is possible through a dedicated webservice for job modification.

The request URL is the same as the URL for viewing the job XML, but where `.analysis.xml` is replaced with `.modify`:

```
POST /DataCleaner-monitor/repository/{tenant}/jobs/{job}.modify
```

The POST request body needs to be a JSON document, specifying the new metadata for the result file. Here's an example:

```
{
    "name":"new name of job",
    "overwrite":true
}
```

### Copying jobs

Copying a job in the DataCleaner monitor repository is possible through a dedicated webservice for job copying.
The request URL is the same as the URL for viewing the job XML, but where '.analysis.xml' is replaced with '.copy':

```
POST /DataCleaner-monitor/repository/{tenant}/jobs/{job}.modify
```

The POST request body needs to be a JSON document, specifying the new metadata for the result file. Here's an example:

```
{
  "name":"name of copy"
}
```

## Deleting jobs

Deleting a job in the DataCleaner monitor repository is possible through a dedicated webservice for job deletion.

The request URL is the same as the URL for viewing the job XML, but where '.analysis.xml' is replaced with '.delete':

Note also that *ALL RELATED* results, timelines and schedules will be deleted, when a job is deleted.

```
POST /DataCleaner-monitor/repository/{tenant}/jobs/{job}.delete
```

The POST request takes no parameters or body information.
Chapter 17. Metric web services

Abstract

Accessing data quality metrics using web services is also possible using DataCleaner monitor. In this chapter we will explain how.

Metrics background

Data quality metrics are all contained within result files. Depending on the job that produced the result, the result may contain different metrics. For instance, if you have a Value distribution analyzer in the job, your result will expose a "Value count" metric (and many more).

To further complicate things, some metrics are parameterized. A metric can be parameterized by either a column name or a query string. For instance, some analyzers (like the String analyzer or Number analyzer) record metrics for multiple columns at the same time. To retrieve metrics from these analyzers you need to provide a column name as a parameter. Other metrics like the "Value count" metric of a Value distribution require a string parameter, specifying which value to get the count of.

Getting a list of available metrics

To get a list of metrics exposed by a particular result, use the following URL pattern:

GET /DataCleaner-monitor/repository/DC/results/[resultname].metrics

Here's an example result:

[  
  {"children":null,"analyzerDescriptorName":"Completeness analyzer","metricDescriptorName":"Invalid row count","analyzerName":null,"analyzerInputName":null,"parameterizedByColumnName":false,"parameterizedByQueryString":false,"metricDisplayName":null,"metricColor":null,"formula":null,"paramColumnName":null,"paramQueryString":null}  
]

From the metrics list we can see that the result carries 3 metrics of a Completeness analyzer. This analyzer only exposes a few basic metrics, which makes it a nice example, but most other analyzers do expose a lot more.

Getting the values of particular metrics

Using the available metrics we saw before, we can build our own list of metrics of interest. This same list can be sent to the same URL with a HTTP POST request:
POST /DataCleaner-monitor/repository/DC/results/[resultname].metrics

As the request body we need to provide the list of metrics that we're interested in. Null values can be left out. For instance:

[  
  {"analyzerDescriptorName":"Completeness analyzer","metricDescriptorName":"Valid row count"},  
  {"analyzerDescriptorName":"Completeness analyzer","metricDescriptorName":"Invalid row count"}  
]

This will return a response like this:

{
  "metricDate":1346154850730,
  "metricValues":{
    "displayName":"Valid row count","value":123},
    {"displayName":"Invalid row count","value":19}]
  
}
Chapter 18. Atomic transformations (data cleaning as a service)

Abstract

In this chapter we will learn how to use the DataCleaner monitor as a web service for invoking DataCleaner jobs as on-demand / on-the-fly data cleansing operations. Like in the case of repository navigation, these web services are located with URLs starting with the form:

/DataCleaner-monitor/repository/DC/...

Where 'DC' is the tenant identifier and 'DataCleaner-monitor' is the web application archive name.

What are atomic transformation services?

DataCleaner jobs normally operate on batches of records, for example all records in a CSV file. However, any DataCleaner job can also be used to process single records on the fly. This feature is called "atomic transformations". Such transformations could be used as a part of a data processing pipeline in a SOA architecture or in a web application with flexible data processing needs.

Creating an atomic transformation is easy. Any DataCleaner job can be used for performing atomic transformations. Simply create a job that does some kind of transformation (anything from a simple 'String length' operation to a complete chain of 'Address cleansing' and regex parsing), and place the job in the DataCleaner monitor repository. A specialized web service endpoint will be automatically exposed to use the job as an atomic transformation. The endpoint URL will be:

/DataCleaner-monitor/repository/DC/jobs/[jobname].invoke

Invoking atomic transformations

The contract of the atomic transformation invocation service is dynamic, based on the contract defined by source data and transformed data in the job.

The web service is based on JSON data. You need to provide the JSON data corresponding to the source record format of the job.

For instance, let's say a job defines the following JavaScript transformation with two source columns (POSTALCODE and COUNTRY):
The web service will therefore expect that each JSON record contains two values. The service accepts one or multiple records in its payload. Use a HTTP POST with a body like this (3 records and 2 columns):

```json
{"rows": [
    {
      "values": ["2200", "DK"],
    },
    {
      "values": ["2200", ""],
    },
    {
      "values": ["DK2200", "DK"],
    }
]}
```

And the resulting response body will look like this:
Atomic transformations
(data cleaning as a service)

```json
{
    "rows": [
        {
            "values": ["DK2200"]
        },
        {
            "values": ["2200"]
        },
        {
            "values": ["DK2200"]
        }
    ],
    "columns": [
        "Country postalcode"
    ]
}
```

Note that the 'Content-Type' header of the request is assumed to be 'application/json'.

Part VII. Invoking DataCleaner jobs from the command-line
**Table of Contents**

19. Command-line interface ................................................................. 78  
   Executables ................................................................................. 78  
   Usage scenarios ......................................................................... 78  
   Executing an analysis job ............................................................. 79  
   Listing datastore contents and available components ...................... 79  
   Parameterizable jobs ................................................................... 81  
   Dynamically overriding configuration elements .............................. 82
Chapter 19. Command-line interface

Abstract

DataCleaner offers a Command-Line Interface (CLI) for performing various tasks, including executing analysis jobs, via simple commands that can be invoked eg. as a scheduled tasks.

Executables

Depending on your distribution of DataCleaner, you will have one of these CLI executables included:

1. `datacleaner-console.exe`, which is a Windows-only executable.
2. `datacleaner.cmd`, which is a script to start DataCleaner in Windows.
3. `datacleaner.sh`, which is a script to start DataCleaner in Unix-like systems, like Linux and Mac OS.
4. If you're running DataCleaner in Java Webstart mode, then there is no Command-Line Interface!

Usage scenarios

The usage scenarios of DataCleaner’s CLI are:

1. Executing an analysis job
2. List registered datastores
3. List schemas in a datastore
4. List tables in a schema
5. List columns in a table
6. List available analyzers, transformers or filters

How these scenarios are attained is revealed by invoking your executable with the `-usage` argument:

```bash
> datacleaner-console.exe -usage
-conf (-configuration, --configuration-file) FILE
: XML file describing the configuration of AnalyzerBeans
-ds (-datastore, --datastore-name) VAL
: Name of datastore when printing a list of schemas, tables or columns
-job (--job-file) FILE
: An analysis job XML file to execute
-list [ANALYZERS | TRANSFORMERS | FILTERS | DATASTORES | SCHEMAS | TABLES | COLUMNS]
: Used to print a list of various elements available in the configuration
-s (-schema, --schema-name) VAL
: Name of schema when printing a list of tables or columns
-t (-table, --table-name) VAL
: Name of table when printing a list of columns
```
Executing an analysis job

Here's how to execute an analysis job - we'll use the bundled example job "employees.analysis.xml":

> datacleaner-console.exe -job examples/employees.analysis.xml
SUCCESS!

...

RESULT:
Value distribution for column: REPORTSTO
Top values:
- 1102: 6
- 1143: 6
- 1088: 5
Null count: 0
Unique values: 0

RESULT:
<table>
<thead>
<tr>
<th>Match count</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aaaaaaaa</td>
<td>22 William</td>
</tr>
<tr>
<td>Aaaa Aaa</td>
<td>1 Foon Yue</td>
</tr>
</tbody>
</table>

RESULT:
<table>
<thead>
<tr>
<th>Match count</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaaaaaaaaa</td>
<td>23 jfirrelli</td>
</tr>
</tbody>
</table>

RESULT:
<table>
<thead>
<tr>
<th>Match count</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aaaaa Aaa</td>
<td>17 Sales Rep</td>
</tr>
<tr>
<td>AA Aaaaaaaaa</td>
<td>2 VP Marketing</td>
</tr>
<tr>
<td>Aaaa Aaaaaaa (AAAA)</td>
<td>1 Sale Manager (EMEA)</td>
</tr>
<tr>
<td>Aaaaa Aaaaaaa (AA)</td>
<td>1 Sales Manager (NA)</td>
</tr>
<tr>
<td>Aaaa Aaaaaaa (AAAAA, AAA)</td>
<td>1 Sales Manager (JAPAN, APAC)</td>
</tr>
<tr>
<td>Aaaaaaaaaaaa</td>
<td>1 President</td>
</tr>
</tbody>
</table>

...

As you can see from the listing, the results of the analysis will be printed directly to the command-line output. If you want to save the results to a file, simply use your operating systems built-in functionality to pipe command-line output to a file, typically using the '>' operator.

Listing datastore contents and available components

The Command-Line Interface allows for listing of datastore contents and available components. The intended usage for this is to aid in hand-editing an analysis file, if this is desired. By using the -list
arguments you can get the metadata of your datastore and the DataCleaner components that will allow you to manually compose an analysis file.

Listing the contents of a datastore is pretty self-explanatory, if you look at the output of the `-usage` command. Here’s a few examples, using the example database `orderdb`:

```bash
> datacleaner-console.exe -list datastores
Datastores:
-----------
Country codes
orderdb

> datacleaner-console.exe -list tables -ds orderdb
Tables:
-------
CUSTOMERS
CUSTOMER_W_TER
DEPARTMENT_MANAGERS
DIM_TIME
EMPLOYEES
OFFICES
ORDERDETAILS
ORDERFACT
ORDERS
PAYMENTS
PRODUCTS
QUADRANT_ACTUALS
TRIAL_BALANCE

> datacleaner-console.exe -list columns -ds orderdb -table employees
Columns:
--------
EMPLOYEENUMBER
LASTNAME
FIRSTNAME
EXTENSION
EMAIL
OFFICECODE
REPORTSTO
JOBTITLE
```

Listing DataCleaner’s components is done by setting the `-list` argument to one of the three component types: ANALYZER, TRANSFORMER or FILTER:

```bash
> datacleaner-console.exe -list analyzers
...
name: Matching analyzer
- Consumes multiple input columns (type: UNDEFINED)
- Property: name=Dictionaries, type=Dictionary, required=false
- Property: name=String patterns, type=StringPattern, required=false
```
name: Pattern finder
- Consumes 2 named inputs
  Input column: Column (type: STRING)
  Input column: Group column (type: STRING)
- Property: name=Discriminate text case, type=Boolean, required=false
- Property: name=Discriminate negative numbers, type=Boolean, required=false
- Property: name=Discriminate decimals, type=Boolean, required=false
- Property: name=Enable mixed tokens, type=Boolean, required=false
- Property: name=Ignore repeated spaces, type=Boolean, required=false
- Property: name=Upper case patterns expand in size, type=boolean, required=false
- Property: name=Lower case patterns expand in size, type=boolean, required=false
- Property: name=Predefined token name, type=String, required=false
- Property: name=Predefined token regexes, type=String, required=false
- Property: name=Decimal separator, type=Character, required=false
- Property: name=Thousands separator, type=Character, required=false
- Property: name=Minus sign, type=Character, required=false

... datacleaner-console.exe -list transformers

... name: Tokenizer
- Consumes a single input column (type: STRING)
- Property: name=Delimiters, type=char, required=true
- Property: name=Number of tokens, type=Integer, required=true
- Output type is: STRING
name: Whitespace trimmer
- Consumes multiple input columns (type: STRING)
- Property: name=Trim left, type=boolean, required=true
- Property: name=Trim right, type=boolean, required=true
- Property: name=Trim multiple to single space, type=boolean, required=true
- Output type is: STRING

... Parameterizable jobs

If you want to make a part of a job parameterizable/variable, then it is possible to do so. Currently this is a feature only supported by means of editing the .analysis.xml files though, since the DataCleaner graphical user interface does not store job variables when saving jobs.

In the source section of your job, you can add variables which are key/value pairs that will be referenced throughout your job. Each variable can have a default value which will be used in case the variable value is not specified. Here's a simple example:

...

<source>
  <data-context ref="my_datastore" />
  <columns>
In the example we've defined two variables: `filename` and `separator`. These we can refer to for specific property values, further down in our job:

```xml
...<analyzer>
  <descriptor ref="Write to CSV file"/>
  <properties>
    <property name="File" ref="filename"/>
    <property name="Quote char" value="&quot;"/>
    <property name="Separator char" ref="separator"/>
  </properties>
  <input ref="col_1"/>
  <input ref="col_2"/>
</analyzer>...
```

Now the property values of the `File` and `Separator char` properties in the `Write to CSV file` have been made parameterizable. To execute the job with new variable values, use `-var` parameters from the command line, like this:

```
DataCleaner-console.exe -job my_job.analysis.xml -var filename=output/my_file.csv -var separator=;
```

### Dynamically overriding configuration elements

Since version 2.5 of DataCleaner it is possible to override elements in the configuration file dynamically from the command line. This is a feature which can be useful in scenarios where you want to invoke the same job but with slightly different configuration details.

For example, you might want to reuse the same job to be executed on several similar CSV files, or similar database environments. Let us assume that you have a CSV datastore that is defined like this:

```xml
<csv-datastore name="My csv file">
  <filename>/path/to/file.csv</filename>
</csv-datastore>
```
To override the filename dynamically, you have to specify the property path (datastore catalog, then datastore name, then property name) with a ‘-D’ parameter on the command line. Furthermore any spaces or dashes are removed and the succeeding character is uppercased. In the end it will look like "camelCase" strings, like this:

DataCleaner-console.exe ... -DdatastoreCatalog.myCsvFile.filename=anotherfile.csv

This mechanism can be used for any configuration property within the datastore catalog and reference data catalog.
Part VIII. Third party integrations
# Table of Contents

20. Pentaho integration ........................................................................................................... 86  
   Launch DataCleaner to profile Pentaho Data Integration steps ........................................ 86  
   Run Pentaho Data Integration jobs in DataCleaner monitor ........................................... 86  
   Run DataCleaner jobs in Pentaho Data Integration ......................................................... 87
Chapter 20. Pentaho integration

Abstract

DataCleaner offers a number of integrations with the Pentaho [http://www.pentaho.com] open source business intelligence suite. In this chapter we will present an overview of the options available.

Launch DataCleaner to profile Pentaho Data Integration steps

In Pentaho Data Integration you can launch DataCleaner by right-clicking any step of your transformations. This will start up DataCleaner with the transformations data pre-loaded, ready for profiling.

This functionality requires installation of the data profiling plugin for Pentaho Data Integration. The instructions and further documentation of this is maintained at Pentaho's wiki page: Kettle Data Profiling with DataCleaner [http://wiki.pentaho.com/display/EAI/Kettle+Data+Profiling+with+DataCleaner].

Run Pentaho Data Integration jobs in DataCleaner monitor

DataCleaner monitor can also be used to run and schedule other jobs than just DataCleaner jobs. An example of this is using DataCleaner monitor to schedule a Pentaho Data Integration transformation. The interoperability in this case is provided through the Carte service provided by Pentaho Data Integration.

Tip

For more information on configuring Carte, we recommend the Pentaho wiki page on the subject: http://wiki.pentaho.com/display/EAI/Carte+Configuration

To set up the job in DataCleaner monitor, go to the 'Scheduling' page and click 'Build job'. Select the Pentaho Data Integration transformation option:
Follow the on-screen wizard to connect to the Pentaho Carte service and select the transformation to execute.

When the job has been registered, you can use DataCleaner monitor to schedule it, and to monitor execution metrics exposed by the job on the 'Dashboard' page (read more in the section about metric charting).

**Run DataCleaner jobs in Pentaho Data Integration**

_Pentaho Data Integration job entry_. If you want to have DataCleaner scheduled and integrated into an environment where you can eg. iterate over files in a folder etc., then you can use Pentaho Data Integration (PDI), which is an open source ETL tool that includes a scheduler.

Construct a PDI "job" (ie. not a "transformation") and add the DataCleaner job entry. The configuration dialog will look like this:
Pentaho integration

The most tricky part is to fill out the executable and the job filename. Note that all configuration options can contain PDI variables, like it is the case with `${user.home}` in the screenshot above. This is useful if you want to eg. timestamp your resulting files etc.
Part IX. Developer's guide
Table of Contents

21. Architecture ......................................................................................................................... 91
    Data access ............................................................................................................................. 91
    Processing framework ............................................................................................................. 91
22. Executing jobs through code ............................................................................................... 93
    Overview of steps and options .............................................................................................. 93
    Step 1: Configuration ............................................................................................................. 94
    Step 2: Job ............................................................................................................................... 96
    Step 3: Execution ................................................................................................................... 99
    Step 4: Result .......................................................................................................................... 101
23. Developer resources ............................................................................................................. 103
    Extension development tutorials ......................................................................................... 103
    Building DataCleaner ........................................................................................................... 103
24. Extension packaging ............................................................................................................. 104
    Annotated components ......................................................................................................... 104
    Single JAR file ....................................................................................................................... 104
    Extension metadata XML ...................................................................................................... 104
    Component icons .................................................................................................................. 105
25. Embedding DataCleaner ....................................................................................................... 106
Chapter 21. Architecture

Abstract

The architecture of DataCleaner can be described from different angles depending on the topic of interest. In the following sections we cover different aspects of the DataCleaner architecture.

Data access

In DataCleaner all sources of data are called 'datastores'. This concept covers both sources that are read/parsed locally and those that are 'connected to', eg. databases and applications. Some datastores can also be written to, for instance relational databases.

DataCleaner uses the Apache MetaModel framework [http://metamodel.incubator.apache.org] for data access. From DataCleaner's perspective, MetaModel provides a number of features:

1. A common way of interacting with different datastores.
2. A programmatic query syntax that abstracts away database-specific SQL dialects, and that is usable also for non-SQL oriented datastores (files etc.).
3. Out-of-the-box connectivity to a lot of sources, eg. CSV files, relational databases, Excel spreadsheets and a lot more.
4. A framework for modelling new sources using the same common model.

DataCleaners datastore model is also extensible in the way that you can yourself implement new datastores in order to hook up DataCleaner to legacy systems, application interfaces and more. For more information refer to the Developer resources chapter.

Processing framework

For processing data, DataCleaner builds upon the AnalyzerBeans framework [http://analyzerbeans.eobjects.org] framework, which is a closely related, but independent project. The goal of AnalyzerBeans is to provide a performant and extensible framework for batch-analyzing data.

AnalyzerBeans is different from most conventional data processing frameworks in that you do not specify a specific order of events, but the framework infers it. If a step (aka. a component) in the flow requires input from another step, then the flow is automatically layed out so that the dependent step is executed before the other. Please refer to the section on wiring components together for an walkthrough on how this works in practice.

One of the key characteristics of AnalyzerBeans is that it is built for massively parallel processing. Whereas most data processing frameworks archive parallelism by the use of parallel execution of each step in a flow, the AnalyzerBeans framework executes the processing of each record in parallel. This means that any CPU utilization is drastically improved since unequal processing time of the steps in a processing chain does not cause in bottlenecks! The downside to this approach is that the order of the processed records cannot be guaranteed, but this is only very rarely required in the domain of data profiling and analysis, and if it is required there are technical workarounds to apply.

To further optimize performance, the AnalyzerBeans framework also allows certain components to modify the source query of the processing chain! This is a mechanism known as "push down query optimization", and is only applicable in scenarios where all succeeding components depend on a specific condition.
But in those cases (for instance a filter specifying that a field is NOT NULL), then it can drastically improve the time needed to process all records. For more information on this principle, please read the blog entry 'Push down query optimization in DataCleaner [http://kasper.eobjects.org/2011/12/push-down-query-optimization-in.html]' by Kasper Sørensen.
Chapter 22. Executing jobs through code

Abstract

In this chapter we will go through one of the most common use-cases for integrating DataCleaner into your own application; Executing a DataCleaner job through code.

Overview of steps and options

There's a couple of variants of this story - What kind of configuration options do you want? Would you like to build the job programatically, or have it somewhere on disk as a .analysis.xml file? Will you be doing any processing of the result, or will the job itself contain all the necessary logic.

The various steps and options are depicted in the diagram below. In the following sections we'll go through each of the 4 steps/columns in the diagram:
Step 1: Configuration

1. Load from file `conf.xml`
2. Use JaxbConfigurationReader to parse configuration file
3. Build programmatically `AnalyzerBeansConfiguration`
4. Use `AnalyzerBeansConfigurationImpl` (important: Datastores + TaskRunner)
Executing jobs through code

The configuration for DataCleaner is represented in the class `AnalyzerBeansConfiguration`, named after the underlying processing framework called `AnalyzerBeans`. You need a `AnalyzerBeansConfiguration` as a prerequisite for most of the coming operations.

The easiest and probably most convenient option for acquiring an `AnalyzerBeansConfiguration` instance is to load it from a file, typically named `conf.xml` (See the Configuration file chapter for more details on this file format). To load the file, use the `JaxbConfigurationReader` class, like this:

```java
InputStream inputStream = new FileInputStream("conf.xml");
JaxbConfigurationReader configurationReader = new JaxbConfigurationReader();
AnalyzerBeansConfiguration configuration = configurationReader.read(inputStream);
```

Alternatively, you can build the configuration programatically, through code. This is typically more cumbersome, but in some cases also quite useful if the configuration is to be build dynamically or something like that.

Here's an example where we configure `AnalyzerBeans` with 2 example datastores and a threadpool of 10 threads:

```java
Datastore datastore1 = new CsvDatastore("my CSV file", "some_data.csv");
boolean multipleConnections = true
Datastore datastore2 = new JdbcDatastore("my database",
    "username", "password", multipleConnections);

AnalyzerBeansConfigurationImpl configuration = new AnalyzerBeansConfigurationImpl();
configuration = configuration.replace(new MultiThreadedTaskRunner(10));
configuration = configuration.replace(new DatastoreCatalogImpl(datastore1, datastore2));
```

Either way we do it, we now have an `AnalyzerBeansConfiguration` with the variable name 'configuration'. Then we can proceed to defining the job to run.
Step 2: Job

Like with the configuration, we can choose to either load the job we want to run from a file, or build it programmatically.

Let's start by simply loading a job from a file. We'll need to use the JaxbJobReader class:

```java
InputStream inputStream = new FileInputStream("my_job.analysis.xml");
JaxbJobReader jobReader = new JaxbJobReader(configuration);
AnalysisJob analysisJob = jobReader.read(inputStream);
```

Note that this is the 'vanilla' case. You can also use the JaxbJobReader to read metadata about a job, and even to read a job 'as a template', which makes it possible to instantiate the job with certain replacements. For an example of how this functionality is used in DataCleaner's desktop application, see the template jobs section.
The other way of producing a job is to build it programmatically. This is quite involved process that varies quite a lot depending on what kind of job you want to build. But the API has been designed to make it as easy as possible.

To give an overview of the API, consider this list of important classes:

1. **AnalysisJobBuilder**: Represents a mutable job that is being built. This builder object contains source columns of the job, and all the components that consume source columns (or sometimes transformed columns).

2. **TransformerJobBuilder**, **FilterJobBuilder**, and **AnalyzerJobBuilder**: represents mutable components of the job that is being built. These can each have configuration properties, filter requirements, input and output columns.

**Tip**

Be aware of the unfortunate similarity between the 'AnalyzerJobBuilder' class name and the 'AnalysisJobBuilder' class name. To rid the confusion, remember that the 'analysis' represents the full scope of the job, whereas an 'analyzer' is just a single active part ('component') of the job.

Let's see an example of building a job programmatically. And to ensure that we don't miss important insights, we'll make it a fairly non-trivial job with both filters, transformers and analyzers. The job will encompass:

1. Three source columns from the datastore 'my database': Name, Age and Company_name.
2. All records where 'Company_name' is null will be inserted into the datastore called 'my CSV file'. In the CSV file the columns are called 'fullname' and 'age_years'.
3. All records where 'Company_name' isn't null will 1) have their working address looked up in another table of the database, and 2) the name and the working address will be passed on to a 'Pattern finder' analyzer.

```java
Datastore myDatabase = configuration.getDatastoreCatalog().getDatastore("my database");
Datastore myCsvFile = configuration.getDatastoreCatalog().getDatastore("my CSV file");
AnalysisJobBuilder builder = new AnalysisJobBuilder(configuration);
builder.setDatastore(myDatabase);
builder.addSourceColumns("public.persons.Name","public.persons.Age","public.persons.Company_name")

InputColumn<?> nameColumn = builder.getSourceColumnByName("Name");
InputColumn<?> ageColumn = builder.getSourceColumnByName("Age");
InputColumn<?> companyColumn = builder.getSourceColumnByName("Company_name");

// add a filter to check for null 'company'
FilterJobBuilder<NullCheckFilter> nullCheckBuilder = builder.addFilter(NullCheckFilter.class);
nullCheckBuilder.addInputColumn(companyColumn);

// add a InsertIntoTable analyzer to write the records without a company to the 'my CSV file'
AnalyzerJobBuilder<InsertIntoTableAnalyzer> insertBuilder = builder.addAnalyzer(InsertIntoTableAnalyzer.class);
insertBuilder.addInputColumns(nameColumn, ageColumn);
insertBuilder.setConfiguredProperty("Datastore", myCsvFile);
insertBuilder.setConfiguredProperty("Columns", new String[] {"fullname","age_years"});
insertBuilder.setRequirement(nullCheckBuilder.getOutcome(NullCheckFilter.Category.NULL));
```
Executing jobs through code

// add a lookup for the company working address
TransformerJobBuilder<TableLookupTransformer> lookupBuilder =
  builder.addTransformer(TableLookupTransformer.class);
lookupBuilder.addInputColumn(companyColumn);
lookupBuilder.setConfiguredProperty("Datastore", myDatabase);
lookupBuilder.setConfiguredProperty("Schema name", "public");
lookupBuilder.setConfiguredProperty("Table name", "companies");
lookupBuilder.setConfiguredProperty("Condition columns", new String[] {"name"});
lookupBuilder.setConfiguredProperty("Output columns", new String[] {"address"});
lookupBuilder.setRequirement(nullCheckBuilder.getOutcome(NullCheckFilter.Category.NOT_NULL));

// reference the 'working address' column and give it a proper name
MutableInputColumn<?> addressColumn = lookupBuilder.getOutputColumns().get(0);
addressColumn.setName("Working address");

// add the Pattern finder analyzer
PatternFinder patternFinder = jobBuilder.addAnalyzer(PatternFinder.class);
patternFinder.addInputColumns(nameColumn, addressColumn);

// validate and produce to AnalysisJob
AnalysisJob analysisJob = jobBuilder.toAnalysisJob();

Things to note from this example:
1. Notice how the filter requirements are set up using the .setRequirement(...) method on the succeeding components.

2. There aren't any explicit filter requirements set on the 'Pattern finder' analyzer. This isn't necessary since it depends on a transformed input column ('Working address') which itself has the requirement. DataCleaner will figure out the transitive requirements automatically.

3. One piece of 'magic' is how to set the properties of the components correctly. We can see that we call .setConfiguredProperty(String, Object), but not how to figure out what to pass as arguments. There are two proper ways to figure this out...

   a. You can use DataCleaner's command line to list all components of a specific type, e.g.:

        > DataCleaner-console.exe -list ANALYZERS

        ... name: Insert into table
        - Consumes 2 named inputs
          Input columns: Additional error log values (type: Object)
          Input columns: Values (type: Object)
        - Property: name=Column names, type=String, required=true
        - Property: name=Datastore, type=UpdateableDatastore, required=true

   b. Or you can simply open up the component class in an IDE to inspect it's @Configured properties. For instance, if we look at InsertIntoTableAnalyzer.java we'll see:

        ...


@Inject
@Configured
@Description("Names of columns in the target table.")
@ColumnProperty
String[] columnNames;

@Inject
@Configured
@Description("Datastore to write to")
UpdateableDatastore datastore;

... 

From these fields we can infer that there will be two configured properties, 'Column names' and 'Datastore'.

Either way we do it, we now have an AnalysisJob with the variable name 'analysisJob'. Then we can proceed to actually executing the job.

**Step 3: Execution**

Executing the job is one of the easiest steps, but obviously there are options available beyond the 'vanilla' scenario.
Executing jobs through code

The simple scenario of running the job is to use the plain AnalysisRunnerImpl class, like this:

```java
AnalysisRunner runner = new AnalysisRunnerImpl(configuration);
AnalysisResultFuture resultFuture = runner.run(analysisJob);
```

This will return a `AnalysisResultFuture`, which under most circumstances represents a still-running job. Your application can continue to do other work in the background, or it can decide to block by calling `.await()`.

Here's a typical example of handling the result future:

```java
// block until the job has finished
resultFuture.await();

if (resultFuture.isSuccessful()) {
    // do something with the successful result
    handleResult(resultFuture);
} else {
    List<Throwable> errors = resultFuture.getErrors();
    for (Throwable error : errors) {
        logger.error("An error occurred while executing job", error);
    }
    // usually the first error that occurred is the culprit, so we'll throw that
    throw errors.get(0);
}
```

You might ask what kind of errors will happen while executing a DataCleaner job? The answer is that it can be a lot of things, for instance:

1. The connection to the source database or resource may fail somehow.
2. One of the components in the job may throw an unexpected exception.
3. One of the components may throw an exception because it's configuration is incomplete or invalid (although this will in most cases be detected while building the AnalysisJob instance).
4. If you're writing data to another datastore, that may also fail for whatever datastore-dependent reasons.
5. If your job is doing something stupid like a Value Distribution of a billion unique IDs, then you'll run out of memory.

Let's now assume that your job has executed successfully. We'll now look at how you can post-process results and how to save/load them to/from a file.
Great, now we have an `AnalysisResultFuture`, and we've determined that it was successful. What can we do with it?

The results of each analyzer of the job are available through the 'AnalysisResult' interface, which `AnalysisResultFuture` implements. Note that the analyzer result types are very different from each other. For instance, the 'Insert into table' analyzer produces a `WriteDataResult`, while the 'Pattern finder' produces a `PatternFinderResult`. Let's see how you can extract information from them:

```java
// demonstrate the the result
// future implements the AnalysisResult interface, which is sufficient
// for all the followin operations
AnalysisResult analysisResult = resultFuture;
```
List<AnalyzerResult> results = analysisResult.getResults();
for (AnalyzerResult result : results) {
    if (result instanceof WriteDataResult) {
        WriteDataResult writeDataResult = (WriteDataResult)result;
        System.out.println("Inserted " + writeDataResult.getWrittenRowCount() + " records");
    }
    if (result instanceof PatternFinderResult) {
        PatternFinderResult patternFinderResult = (PatternFinderResult)result;
        int matches = patternFinderResult.getMatchCount("Aaaaa Aaaaa");
        int total = patternFinderResult.getTotalCount();
        System.out.println("There were " + matches + " matches out of " + total + " for our standard pattern.");
    }
}

As you can see, how you handle the result depends a lot on what type of result is produced.

For generic handling of results, including all the possible result extensions that might occur, DataCleaner employs a renderer framework which selects a result renderer according to type and precedence. If you need such generic functionality, take a look at the classes RendererBean, RendererFactory, Renderer and RenderingFormat.

One common requirement is to persisting it. We recommend doing this by means of Java's serialization, since analysis results are polymorphic and it's structure may be dependent on extensions. You can also device a more "structured" persistance scheme, but beware that it will require quite some stability in terms of which analyzers you add to your jobs.

So let's see how we use Java serialization. But unfortunately AnalysisResultFuture isn't serializable! There is however a class which shares the interface 'AnalysisResult' with 'AnalysisResultFuture', that is serializable, Namely 'SimpleAnalysisResult'. Let's see how to use it and serialize our result to a .analysis.result.dat file, (which DataCleaner can read):

    // make the result serializeable
    AnalysisResult analysisResult = resultFuture;
    analysisResult = new SimpleAnalysisResult(analysisResult.getResultMap());
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("my_result.analysis.result.dat"));
    oos.writeObject(analysisResult);
    oos.close();

And now let's, for example sake, also load our file by deserializing it. For this we need to use the ChangeAwareObjectInputStream class, which ensures backwards compatible deserialization of objects:

    ObjectInputStream ois = new ChangeAwareObjectInputStream(new FileInputStream("my_result.analysis.result.dat"));
    AnalysisResult analysisResult = (AnalysisResult) ois.readObject();

Now the result is restored and you can further work with it.
Chapter 23. Developer resources

Extension development tutorials

There are many useful resources for those who engage in developing extensions (aka. plugins / add-ons) to DataCleaner. To help you on your way, here's a list of useful links. If you think this list is missing a link, please let us know:


Building DataCleaner

Get the source code for AnalyzerBeans and DataCleaner from GitHub:

```
> git clone https://github.com/datacleaner/AnalyzerBeans.git AnalyzerBeans
> git clone https://github.com/datacleaner/DataCleaner.git DataCleaner
```

Build the projects:

```
> cd AnalyzerBeans
> mvn clean install
> cd ../DataCleaner
> mvn clean install
```

Run DataCleaner

```
> cd ../DataCleaner/desktop/target
> java -jar DataCleaner-desktop-[version].jar
```
Chapter 24. Extension packaging

Abstract

DataCleaner extensions are packages of added functionality, written in Java. To correctly package an extension, this chapter will walk through the details.

Annotated components

The main principle behind extension discovery in DataCleaner is annotated classes. These are the annotations that will work to activate components within your extension:

1. @AnalyzerBean
2. @TransformerBean
3. @FilterBean
4. @RendererBean

Please refer to the javadoc documentation of these annotations/classes for details on usage.

Single JAR file

The extension must consist of a single JAR file. If you have dependencies other than the libraries provided by the DataCleaner distribution, you need to package these inside your own JAR file. If you’re using Maven for your build, the Maven Assembly Plugin can provide this functionality easily using this snippet in your POM:

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.2.1</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Extension metadata XML

To improve the experience, you can optionally include metadata about the extension in an XML file, bundled within the JAR file itself.

The name of the extension metadata file has to be `datacleaner-extension.xml` and be placed in the root directory of the JAR file. Here’s an example of how the file looks like:
<extension xmlns="http://eobjects.org/datacleaner/extension/1.0">
  <name>My extension</name>
  <package>path.to.extension</package>
  <description>This is an example extension. I should put a short description here.</description>
  <icon>path/to/extension/ExtensionIcon.png</icon>
  <author>John Doe</author>
  <url>http://datacleaner.org/extensions</url>
  <version>1.0</version>
</extension>

The added value of this metadata is that DataCleaner can expose this information to the user and also use it to manage updates of the extension etc. The metadata file is however, completely optional.

**Component icons**

If you wish to add a custom icon for your components (e.g., a transformer or analyzer), you need to place the icon as a PNG image with the same name as the fully classified class name of the component.

An example: If your component class name is "com.company.ext.MyAnalyzer", then the icon for this component should be located at "/com/company/ext/MyAnalyzer.png" in the extension JAR file.

Similarly, if you bundle your own ComponentCategory implementations (which define the menu groups in DataCleaner), you can define icons for these by adding a PNG file with a fully classified filename corresponding to the ComponentCategory class name.
Chapter 25. Embedding DataCleaner

It is possible to embed DataCleaner into other Java applications. This allows a simple way to add Data Quality Analysis (DQA) and Data Profiling functionality as an addition to the applications that you are building.

The simplest way to embed DataCleaner is simply by doing what DataCleaner's main executable does - instantiate the `Bootstrap` class with default arguments:

```java
BootstrapOptions bootstrapOptions = new DefaultBootstrapOptions(args);
Bootstrap bootstrap = new Bootstrap(bootstrapOptions);
bootstrap.run();
```

To customize further, add your own implementation of the `BootstrapOptions` class. The main scenario for embedding DataCleaner is to run the application in the so-called "single datastore mode". This can be achieved by implementing the `BootstrapOptions` and providing a non-null value for the `getSingleDatastore()` method.