Endeca® Platform Services

Control System Guide

Version 6.1.1 • December 2011

Contents

Copyright and disclaimer

Preface

Endeca® InFront enables businesses to deliver targeted experiences for any customer, every time, in any channel. Utilizing all underlying product data and content, businesses are able to influence customer behavior regardless of where or how customers choose to engage — online, in-store, or on-the-go. And with integrated analytics and agile business-user tools, InFront solutions help businesses adapt to changing market needs, influence customer behavior across channels, and dynamically manage a relevant and targeted experience for every customer, every time.

InFront Workbench with Experience Manager provides a single, flexible platform to create, deliver, and manage content-rich, multichannel customer experiences. Experience Manager allows non-technical users to control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

At the core of InFront is the Endeca MDEX Engine,™ a hybrid search-analytical database specifically designed for high-performance exploration and discovery. InFront Integrator provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. InFront Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

These components — along with additional modules for SEO, Social, and Mobile channel support — make up the core of Endeca InFront, a customer experience management platform focused on delivering the most relevant, targeted, and optimized experience for every customer, at every step, across all customer touch points.

About this guide

This guide describes the tasks involved in the configuration and administration of an Endeca implementation running in an Endeca Control System environment.

Although the Control System is deprecated, many existing applications still use it. However, if you are developing a new application, Endeca strongly recommends that you do so using the Endeca Application Controller and not the Control System. The Endeca Application Controller is documented in the *Endeca Application Controller Guide*.

Who should use this guide

This guide is intended for system administrators and others who are managing the day-to-day operations of the Endeca Information Access Platform using the Endeca Control System (Control Interpreter and JCD). It may also be of interest to developers while they are deploying an Endeca implementation.

Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

Contacting Endeca Customer Support

The Endeca Support Center provides registered users with important information regarding Endeca software, implementation questions, product and solution help, training and professional services consultation as well as overall news and updates from Endeca.

You can contact Endeca Standard Customer Support through the Support section of the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

Chapter 1

Endeca Control System Overview

The Endeca Control System provides a way to control and administer your Endeca implementation. This section provides an overview of the Endeca Control System.

About the Endeca Control System

The Endeca Control System includes the Control Interpreter and the Endeca Job Control Daemon (on UNIX) and the Endeca JCD Service (on Windows). These components control and administer the Endeca Information Access Platform (IAP) software running on one or more host machines.

This guide uses the term "Endeca JCD" to refer to the component on either a UNIX or Windows platform. When necessary, platform-specific differences between the components are called out in context.

The Endeca Control System should be installed on all machines that host the Endeca software.

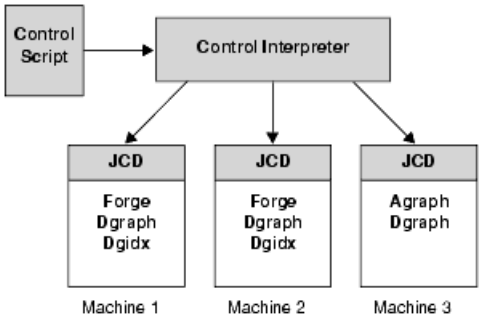
The Endeca JCD provides the Endeca Information Access Platform with reliable process execution and lightweight job management. The Endeca JCD runs on every machine in an implementation, and is responsible for:

- Executing various Endeca jobs across all hosts in an implementation Examples include fetching data and running the Forge, Dgidx, Dgraph, and Agraph programs.
- Monitoring Endeca jobs and restarting them in case of failure (if the job's settings indicate that it should do so).

An important secondary function of the JCD is to make your Endeca system more resilient to interruptions in service, particularly in a production environment.

The Control Interpreter connects to all of the Endeca JCDs in your deployment so you can coordinate their activities from a single interface. It processes simple scripts contained in a control script file. The scripts tell the Control Interpreter how to use each machine's Endeca JCD to run jobs on that machine. In contrast to the JCD, which runs on every machine, the Control Interpreter runs on only one machine in a deployment.

The following diagram shows the architecture of a typical Endeca Control System:



About the Endeca JCD

Endeca jobs are created as children under the Endeca JCD, so that the JCD can monitor them.

When a child job terminates, the Endeca JCD restarts it immediately if the job has been set to run as a server process. On UNIX, `inittab` is responsible for keeping the Endeca JCD itself running, and restarting it in case of failure. On Windows, the Windows Service Control Manager performs the same task.

In a typical Windows usage, the Endeca JCD is set to start automatically when you start its host computer. In a typical UNIX usage, the Endeca JCD is started from `inittab` on system start. In either environment, the JCD reads in a configuration file (`jcd.conf`) that defines various connection and security parameters. After reading in the configuration file, the Endeca JCD enters server mode. It executes any jobs that were set to execute at start-up and listens for incoming HTTP requests on a port that is specified in the `jcd.conf` file.

The Endeca JCD may receive two types of HTTP requests:

- Automated requests that are sent directly to the JCD from the Control Interpreter.
- Manual requests that an administrator sends to the JCD via a browser-based interface, using commands issued as URL parameters.

Most of the time, the requests the Endeca JCD receives come from the Control Interpreter. This is the preferable method for controlling your Endeca deployment. Administrators, however, can use the Endeca JCD browser interface to get status information on running jobs, and to perform basic tasks like starting and stopping processes.

If required, JCD authentication for both types of requests is provided by SSL certificates. Remote connections to the Endeca JCD can be made from any machine authorized to do so.

Related Links

Enabling authentication and security on page 19

If your implementation requires it, the Endeca JCD can authenticate the identity of all client requests through the use of SSL certificates. You need to generate a set of certificate files to enable SSL.

About the Control Interpreter

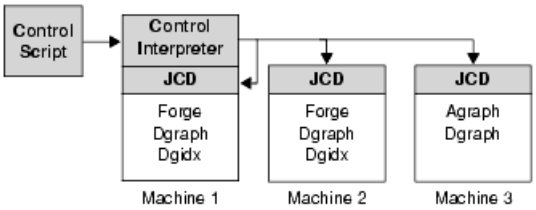
The Control Interpreter calls on the Endeca JCDs to start, stop, and check the status of Endeca server processes (such as the Dgraph and the Agraph), and to run and check the status of arbitrary commands (such as Forge, Dgidx, and other data processing programs).

Note: The remainder of this document refers to both server processes and arbitrary commands as “jobs.”

You can write control scripts for the Control Interpreter that describe an entire data update sequence that runs in parallel on multiple machines. Using control script syntax, you indicate to the Control Interpreter what jobs should be run in sequential order and what jobs should be run in parallel.

The Control Interpreter is not installed by default with the Endeca Platform Services installation. This means that you have to specifically choose to have it installed during the installation process. See the *Endeca Platform Services Installation Guide* for installation instructions.

Even if you install the Control Interpreter on multiple Endeca servers, the control script for your deployment is a custom-made file that is located on only one of those Endeca servers. You cannot run the Control Interpreter without a control script; therefore, it is the server that has the control script file that runs the Control Interpreter software for the entire deployment, as shown below:

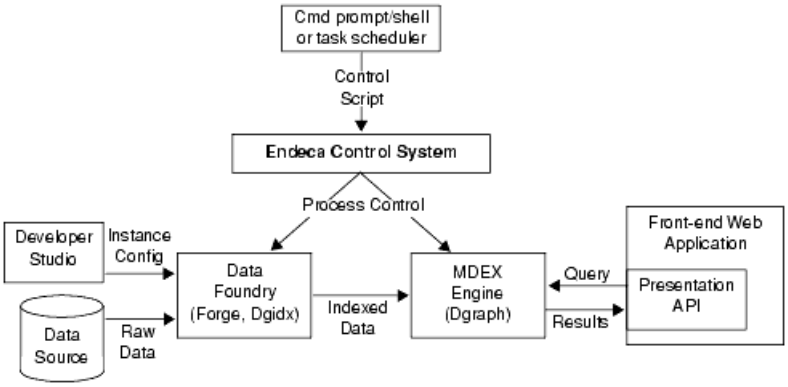


Note: One or more backup control scripts can be placed on other servers in case the first one fails. However, failover is not automatic.

Control Interpreter scripts are executed with detailed logging and monitoring. Error detection and notification are built in, so no fatal error goes unreported.

Control System architecture

The following diagram shows the architecture of a typical implementation that uses the Endeca Control System:



In this architecture diagram, the following happens:

1. The developer creates an instance configuration, using Developer Studio, that determines what data and features will be incorporated into the index.

2. The developer creates a control script that manages all of the resources in the Endeca environment and determines which tasks each machine will perform during a data run.
3. The developer starts the Endeca Control System, either manually or through a task scheduler.
4. The Control System manages the entire data update process, according to the instructions in the control script. This includes running Forge and Dgidx to create indexed data, and starting a Dgraph based on that indexed data.

More detailed information on configuring and using the Endeca Control System is found in later sections of this guide.

Endeca Control System directory structure

Before you start building your instance configuration, you must create a directory structure to support your data processing back end. The structure of the directory is dictated by the mechanism (i.e., Endeca Control System or the Endeca Application Controller) you have chosen to control your Endeca environment.

If you are using the Endeca Control System to control your environment, you will have to create a directory structure to contain source data, control scripts, system-generated files, log files, and so forth. The example below shows the directory structure used for the `sample_wine_data` reference implementation:

```
instance_root
 data
  forge_input
  incoming
  partition0
   dgidx_output
   dgraph_input
   forge_output
   state
 etc
 logs
 reports
```

The table below describes the contents of each directory:

Directory	Description
instance_root	Contains all required subdirectories for this instance of your Endeca implementation.
`data`	Contains subdirectories for your instance configuration, source data extracts, and system-generated files.
`forge_input`	Contains the baseline pipeline file (typically named `pipeline.epx`), the partial updates pipeline file (if you are running partial updates; the file is typically named `partial_pipeline.epx`), and the index configuration files (`*.xml`).

Directory	Description
`incoming`	Contains data ready for processing by Forge. On a production site, the files in this directory may have been created by a data extraction process on the customer's database or may be picked up from another FTP server.
`partition`	Contains subdirectories for system-generated files, such as Forge output, Dgidx output, and Dgraph input.
`state`	Contains any state information that must be saved between runs of the Data Foundry, for example, auto-generated dimension IDs.
`forge_output`	Contains data that has been processed by Forge and is ready for indexing.
`dgidx_output`	Contains indices that have been processed by Dgidx and output in MDEX Engine format.
`dgraph_input`	Contains a copy of the MDEX Engine indices stored in dgidx_output. When you start the MDEX Engine (Dgraph) process, you should point at this copy of the indices. Having a separate copy of the indices allows you to isolate your working MDEX Engine indices from those that are being updated.
`etc`	Contains system-level configuration for your Endeca implementation, such as control scripts.
`logs`	Contains log files generated by the various Endeca components.
`reports`	Contains any reports you choose to generate for your implementation.

While you can structure your directories in any way you want, Endeca recommends you mimic the directory structure of the `sample_wine_data` reference implementation in order to maximize reuse of code, configuration settings, and control scripts.

After creating your directory structure, you should:

- Copy your source data extracts to `instance_root/data/incoming`.
- Copy any control scripts you want to use or modify to the `etc` directory. You can find reference control scripts in the `etc` directory of the `sample_wine_data` reference implementation.

Running a pipeline via the Endeca Control System

After you have created your basic pipeline, you should run it and view the results. Your initial goal is to make sure that your source data is running through the entire pipeline and being incorporated into the MDEX Engine indices.

The Basic Pipeline template does not contain a source data file. Therefore, before you run the Basic Pipeline, make sure you have created an incoming directory that contains source data. Alternatively, you can use the incoming directory in the `sample_wine_data` reference, which contains a source data file named `wine_data.txt.gz`.

For information about the pipeline and its components, see the *Endeca Forge Guide*.

To run your pipeline via the Endeca Control System, you:

1. Write a control script.

   Endeca recommends that, at first, you modify one of the control scripts that comes with the reference implementations, such as the `remote_index.script` located in:

   - `%ENDECA_REFERENCE_DIR%\sample_wine_data\etc` on Windows.
   - `$ENDECA_REFERENCE_DIR/sample_wine_data/etc` on UNIX.

2. Run DefineJobs on your control script to define the jobs that need to be executed by each machine's Endeca JCD.
3. Run RunCommand on your control script to execute the jobs and start the MDEX Engine.

Related Links

The Control Interpreter on page 35
The Control Interpreter manages the activities of multiple Endeca JCD instances, in a distributed Endeca deployment, from a single interface. This section describes how you configure and run the Control Interpreter.

About using control scripts for baseline and partial updates on page 96
This section describes control script development and execution for baseline updates and partial updates.

Pipeline paths in a control script environment

You can use the `--inputDir` flag to specify a base path for your pipeline.

When using a `Pipeline.epx` file in a control scripts environment, pipeline paths for incoming data are treated as follows:

- If you are using the `--inputDir` flag, the path specified with this flag will be used as a base path for the pipeline. This means that if the pipeline specifies a relative path (which can be just a filename), that path will be relative to the base path in the `--inputDir` flag. Note, however, that if the pipeline uses an absolute path, then the path in the `--inputDir` flag is ignored.
- If you are not using the `--inputDir` flag, relative paths are resolved in relation to the location of the control script, while absolute paths are used exactly as specified.

Make sure that any paths are valid and that referenced directories contain the correct data.

Chapter 2

Working With the Endeca JCD

In a UNIX environment, the Endeca Job Control Daemon (JCD) monitors and manages the Endeca Information Access Platform software to provide a robust process execution environment. In a Windows environment, the analogous component is implemented as the Endeca JCD Service. This section discusses the administrator's interaction with the Endeca JCD.

About controlling the Endeca JCD

You can control the Endeca JCD with the `jcd` command.

The executable is located in the following directory:

- `$ENDECA_ROOT/bin` on UNIX
- `%ENDECA_ROOT%\bin` on Windows

Endeca JCD command syntax reference

This reference provides the syntax for using the Endeca JCD command on Windows and UNIX platforms.

On Windows, the Endeca JCD command has the following syntax:

`jcd [--help] [--version] [--register <config-file>]`

On UNIX, the Endeca JCD command has the following syntax:

`jcd [--help] [--version] [<config-file>]`

Descriptions of the options are as follows:

Command	Purpose
`--help`	Displays command-line usage and configuration file settings for the Endeca JCD.
`--version`	Displays the Endeca JCD version information and exits.

Command	Purpose
`--register <config-file>`	Registers the JCD as a Windows service, using the specified Endeca JCD configuration file (for example, `%ENDECA_CONF%\etc\jcd.conf`). You can start the JCD from the Windows Services utility.
`<config-file>`	Starts the JCD on UNIX, using the specified Endeca JCD configuration file (for example, `$ENDECA_CONF/etc/jcd.conf`).

Setting the ENDECA_MDEX_ROOT environment variable

If you have installed the Endeca Control System on a Windows machine, it is recommended that you set *ENDECA_MDEX_ROOT* as a system environment variable, so that the JCD Service uses it when you start the service.

On UNIX, the Endeca Control System (including the Endeca JCD) is installed by default. However, on Windows, the Endeca Platform Services installer does not install the Endeca Control System unless you specifically choose to do so on the **Custom Setup** screen. For details, refer to the *Endeca Platform Services Installation Guide*.

Note: On UNIX systems, after you install the MDEX Engine, you run a script (named `endeca_setup_csh.ini` or `endeca_setup_sh.ini`) to set the *ENDECA_MDEX_ROOT* environment variable.

To set *ENDECA_MDEX_ROOT* as a system environment variable in Windows:

1. Right-click on **My Computer** and then click **Properties**.
2. Click the **Advanced** tab.
3. Click **Environment Variables**.
4. In the **System variables** pane, click **New**.
5. In the **New System Variable** dialog, enter `ENDECA_MDEX_ROOT` as the variable name and the absolute path of the MDEX Engine root directory as the variable value.
   For example, `C:\Endeca\MDEX\6.1.2`.
6. Click **OK**.
7. Click **OK** to exit the **Environment Variables** pane and click **OK** again to exit the **System Properties** dialog.
8. Reboot the system to ensure that the new environment variable is correctly set.
9. Restart the JCD Service from the **Services** pane of the **Computer Management** utility.

The service uses the new *ENDECA_MDEX_ROOT* variable as one of its environment variables.

About starting the Endeca JCD

On Windows, the Endeca JCD starts automatically. On UNIX platforms, it can be started from the command-line.

If you install the Endeca Control System on Windows, the Endeca JCD is registered under the Windows Service Manager and starts up automatically when the operating system boots up. Upon startup, it reads its configuration file (`jcd.conf`), which can include both connection details and security parameters. If the Endeca JCD crashes or is terminated, the Windows Service Manager automatically restarts it.

In a UNIX development environment, the Endeca JCD can be started from the command-line. In a UNIX production environment, however, we recommend that it be started by `init` from `inittab`. If the Endeca JCD crashes or is terminated, `init` automatically restarts it. Upon startup, the Endeca JCD reads its configuration file (`jcd.conf`), which can include both connection details and security parameters.

Note: On UNIX, the Endeca JCD is **not** designed to be run from `inetd`, the Internet services daemon.

Endeca JCD behavior

The following sections describe how the Endeca JCD behaves while it is running and managing jobs.

Important: No job name can contain a dash (`-`) character.

Enabling authentication and security

If your implementation requires it, the Endeca JCD can authenticate the identity of all client requests through the use of SSL certificates. You need to generate a set of certificate files to enable SSL.

The Endeca JCD can authenticate requests made by the Control Interpreter and requests made through the browser-based JCD interface.

Keep in mind that by default, the JCD is not configured to use SSL. Therefore, you must perform all these steps to enable SSL. The procedure is documented in the *Endeca Security Guide*.

In order to use SSL certificates, you must:

1. Run a utility, enecerts, that generates the following set of certificate files. See the *Endeca Security Guide* for more information about using the SSL enecerts utility.
   - `eneCert.pem` — certificate file used by all clients and servers to specify their identity when using SSL. This certificate file should be thought of as the identity of the Endeca system, or as the identity of all components of the Endeca system.
   - `eneCA.pem` — certificate authority file used by all clients and servers to authenticate the other endpoint of a communication channel.
   - `eneCA.key` — private key file that is used by the certificate authority (that is, the enecerts utility) to sign the eneCert.pem certificate.
   - `eneCA.cer` — used by Microsoft Internet Explorer
   - `eneCert.p12` — used by Microsoft Internet Explorer
2. If you have multiple machines in your deployment, copy the certificate files to the same location on all machines.
3. Configure the `jcd.conf` file on all machines:
   a) Configure the JCD to use SSL when communicating with other Endeca components.

b) Specify `eneCert.pem` as the location of the certificate that the JCD should present when communicating with other components.
c) Specify `eneCA.pem` as the location of the certificate authority file the JCD will use to authenticate communication from other components.

4. Configure the Control Interpreter's control script:
   a) Configure the Control Interpreter to use SSL when communicating with the JCD.
   b) Specify `eneCert.pem` as the location of the certificate that the Control Interpreter should present when communicating with the JCD.
5. Import the certificate files into Internet Explorer on each machine from which you want to manually issue Endeca JCD commands.

Note: While this last step is not required to run the Control Interpreter, it is required if you want to connect to the Endeca JCD directly and send it commands via a browser.

Related Links

SSL Configuration for the Control Interpreter on page 107
This appendix describes how to use SSL with the Control Interpreter.

About issuing commands directly to the Endeca JCD on page 27
You can communicate with the JCD using commands issued as URL parameters. The syntax and values are explained below.

About logging and monitoring

The Endeca JCD records informational, warning, and error messages about its operations in a JCD log. Each instance of the Endeca JCD records its logs on its local machine.

The location of the log information is determined by the `log_file` setting in `jcd.conf`. By default, the `jcd.conf` is configured to direct log information to a file in `workspace\logs\JcdLog.txt` (on Windows). If you removed the `log_file` setting from `jcd.conf`, then the Endeca JCD directs log information to the Windows Event Log (for Windows) and the syslog (on UNIX).

Examples of the types of information you will find in an Endeca JCD log include:

- When the Endeca JCD process started.
- What port the Endeca JCD is listening to.
- When the Endeca JCD starts and stops a job.
- When the Endeca JCD auto-restarts a server process job.

The log information that the Endeca JCD produces is JCD-specific, not job-specific. In other words, the JCD log indicates when a job is started, stopped and restarted, but it does not provide detailed information about the job. You configure detailed job logging in the Control Interpreter's control script.

Note: Instead of rolling these logs, the JCD simply checks the output file before starting up the Dgraph (or any other process), and refuses to start that process if the file is already over 1 gigabyte. If the output filename specified is actually a directory, or is read-only, the JCD produces an appropriate error message.

Related Links

Control Interpreter-specific logs on page 48
By default, the Control Interpreter prints out each line of script as it runs. If you need more details, you can specify the `--debug` switch.

Viewing errors in the Windows Event Log

You can check the Windows Event Log for information on errors in the Endeca JCD.

To view errors and other messages in the Windows Event Log:

1. On the machine that has the Endeca JCD log that you want to view, select **Administrative Tools** from the **Windows Control Panel**.
2. Select **Event Viewer**.
3. In the **Event Viewer Tree** pane, select **Application Log**.
4. In the right pane of the **Event Viewer** window, scroll and search the **Source** column to find the event you want to research further.
5. Double-click the event to display the **Event Properties** dialog box, which provides details about the error, warning, or information message.

State management

The Endeca JCD maintains a correct and up-to-date representation of the state of all jobs in its state file. The primary purpose of the state file is to restore a system to its previous state after a crash.

If the Endeca JCD crashes, is stopped, or is terminated, it correctly updates its status information about all jobs under management when subsequently restarted. This means you can stop the Endeca JCD for software maintenance without interrupting managed software processes. For example, if the Endeca JCD is stopped and restarted, any server process that died during the Endeca JCD's downtime will be restarted automatically when the Endeca JCD resumes.

About recovering from job start-up failure

The Endeca JCD will automatically stop trying to start a job if it cannot succeed after a set amount of attempts.

If a job parses properly, but the Endeca JCD cannot start the job after multiple attempts, it records this fact and then stops trying to start the job within one minute. The number of attempts that the Endeca JCD will make is specified in the `jcd.conf` file in the `max_restarts_per minute` setting.

Related Links

Configuration file options on page 22
This table describes the options that can be used in the JCD configuration file.

About configuring the Endeca JCD

The `jcd.conf` settings control how the Endeca JCD itself behaves; for example, what port it listens to and what method it uses to log errors.

The Endeca JCD configuration file, `jcd.conf`, contains two types of information:

- General system settings for the JCD
- Security settings that control access to the JCD

These settings are JCD-specific, not job-specific. By contrast, the definitions for the jobs that an Endeca JCD runs are defined in the Control Interpreter's control script.

Important: JCD settings are defined in the JCD configuration file (`jcd.conf`). Individual job parameters are defined in the Control Interpreter's control script.

Related Links

Defining jobs and running the Control Interpreter on page 36

The Control Interpreter is run based on control scripts, and may require job definitions from the Endeca JCD.

Configuration file options

This table describes the options that can be used in the JCD configuration file.

`port`

Description	HTTP port on which the Endeca JCD listens.
Allowed Values	Any numeric value.
Default	`8088`
Required	Yes

`state`

Description	Pathname for the JCD state file, which is used for automatic back up of information about running Endeca jobs.
Allowed Values	Any string value.
Default	No default value.
Required	Yes

`log`

Description	Log output mode.
Allowed Values	`stderr`, `syslog`, or `file`
Default	stderr
Required	No

log_file

Description	Location of the log file, for `log=file` mode.
Allowed Values	Any string value.
Default	No default value.
Required	Yes, if `log=file`. No, if `log=stderr` or `log=syslog`.

ssl

Description	Set to true if you want the JCD to use secure, encrypted communication (HTTPS) when communicating with other Endeca components. Set to `false` to disable encrypted JCD communication. Endeca recommends that you set to `true` for production environments. However, in a development environment, you may opt to set it to `false` to avoid the need for certificate files during development.  **Note:** If you set `ssl=false` then `sslcertfile` and `sslcafile` are ignored.
Allowed Values	`true` or `false`
Default	`false`
Required	No

sslcertfile

Description	Specifies the path of the certificate file that the JCD should present when communicating with other Endeca components via SSL.
Allowed Values	Pathname to `eneCert.pem`. (This is the name of the certificate file. Note that this file is not provided by default. You must run the enecerts utility to create it, as described in the *Endeca Security Guide*.
Default	`eneCert.pem`
Required	Required (and only used) when `ssl=true`.

sslcafile

Description	Specifies the path of the certificate authority file, `eneCA.pem`, that the JCD should use to authenticate communication with other Endeca components.
Allowed Values	Pathname to `eneCA.pem`. (This is the name of the certificate authority file. Note that this file is not provided by default. You must run the `enecerts` utility to create it, as described in the *Endeca Security Guide*.
Default	`eneCA.pem`
Required	Used only when `ssl=true`.

sslcipher

Description	Sets the cipher string (such as `RC4-SHA`) that specifies the cryptographic algorithm the JCD will use during the SSL negotiation. Used only when `ssl=true`.
Allowed Values	Any string value.
Default	No default value.
Required	No

shutdown_timeout_seconds

Description	Maximum time the Endeca JCD will wait, in seconds, for a job to respond to a stop request before killing it.
Allowed Values	Any numeric value.
Default	`30`
Required	No

max_restarts_per_minute

Description	Maximum number of times, per minute, that the Endeca JCD will attempt to restart a server process. JCD stops trying to start the job after this many attempts if the job fails to run.
Allowed Values	Any numeric value.

Default	`10`
Required	No

`max_read_tries`

Description	Maximum number of times that the Endeca JCD will attempt to read data from an incoming client request.
Allowed Values	Any numeric value.
Default	`32`
Required	No

`max_read_time_seconds`

Description	For an incoming client request, the maximum number of seconds that are allowed without data having being read.
Allowed Values	Any numeric value.
Default	`30`
Required	No

`max_write_time_seconds`

Description	The maximum number of seconds that are allowed per reply without data having being read by the client.
Allowed Values	Any numeric value.
Default	`30`
Required	No

Configuration file example

The following is an example of a Windows Endeca JCD configuration file:

```
# Reference implementation of configuration file for
# the Endeca JCD.
# Copyright (c) 2009 Endeca Technologies, Inc.

# Communication port for Endeca JCD. This must match
# the "jcd_port" value used in the Control Interpreter's
# control script (specified # globally as is here or in
# a Machine brick).
port   8088

# State file path, used for automatic backup of
# information about running Endeca jobs (e.g., data
# processing commands, MDEX Engine, etc.):
state C:\Endeca\PlatformServices\workspace\state\JcdState.dat

# Log configuration. Can be file, stderr, or syslog.
# Syslog directs logs to Windows Event Viewer. File is
# recommended.
log     file
log_file C:\Endeca\PlatformServices\workspace\logs\JcdLog.txt

# Security configuration:
# ssl--Set to true if you want the JCD to use encrypted
# communication when communicating with other Endeca
# components. If set to false, sslcertfile and sslcafile
# are ignored.
# sslcertfile--Specifies the path of the certificate
# file that the JCD should present when communicating
# with other Endeca components via SSL.
# sslcafile--Specifies the path to a certificate
# authority file, if you want the JCD to authenticate
# communications from other Endeca components.
ssl     false
sslcertfile C:\Endeca\PlatformServices\workspace\etc\eneCert.pem
sslcafile C:\Endeca\PlatformServices\workspace\etc\eneCA.pem
```

About sending requests directly to the Endeca JCD

You can connect to the Endeca JCD to check the status of running server processes, and to occasionally control these processes at a low level (for example, to stop a running server process). You communicate with the JCD via a browser-based interface, either by accessing the JCD home page or issuing commands as URL parameters.

If you have set the Endeca JCD to use HTTPS mode with authentication (in the `jcd.conf` file, `ssl=true` and `sslcertfile` and `sslcafile` are specified), you must import the SSL certificates into Internet Explorer in order to communicate with the JCD. Importing certificates is an optional step.

Related Links

About issuing commands directly to the Endeca JCD on page 27

You can communicate with the JCD using commands issued as URL parameters. The syntax and values are explained below.

The Endeca JCD home page reference on page 27

For some Endeca JCD parameters, rather than typing in parameter/value pairs, you can enter the hostname and port number (that is, `http://[host]:[port]`) in order to access the Endeca JCD home page, and then click on one of its options, explained below.

About importing SSL certificates into Internet Explorer on page 33
If you use HTTPS mode with authentication, you will need to import the SSL certificates into Internet Explorer to communicate with the JCD.

The Endeca JCD home page reference

For some Endeca JCD parameters, rather than typing in parameter/value pairs, you can enter the hostname and port number (that is, `http://[host]:[port]`) in order to access the Endeca JCD home page, and then click on one of its options, explained below.

The home page offers the following options:

Option	Details
Help	Lists the URL options for the Endeca JCD.
Display running jobs	Shows the status of any running jobs. The job name and status command are both hyperlinked. Clicking on the job name gives you additional information about that job; clicking the command (for example, **Stop** for a running job) executes that command.
Display all defined jobs	Shows detailed information about all defined jobs in the system, regardless of status. The job name and status command are hyperlinked, as described in “Display running jobs,” above. The paths to `stdout`, `stderr`, and the start directory are also linked. Clicking the link to **stdout** or **stderr** opens the associated file, while clicking **StartDir** takes you to that directory and allows you to drill down to its component files.
Display environment variables	Displays a list of all settings in the current environment.
Browse the file system	Displays files and directories in the system in hyperlinked format so that you can drill down on them.

About issuing commands directly to the Endeca JCD

You can communicate with the JCD using commands issued as URL parameters. The syntax and values are explained below.

The general URL syntax for commands sent manually to the Endeca JCD is as follows:

`http://[host]:[port]/[op]?[param]=[val]&[param]=[val]...`

Note: If you are using HTTPS mode, use `https` instead of `http` in the URL.

Alternatively, you can access some of the parameters through the Endeca JCD home page.

Valid *[op]* values are listed below, along with associated *[param]* keys. Note that some of the operations require a job name. Job names are defined in the Control Interpreter's control script.

Note: Due to the length of the URLs some of the examples in this section break onto additional lines, but you would type them on a single line in your Web browser's **Address bar**.

`active`

Description	Gets a brief status for a job. Returns the status for all running jobs if no job parameter is specified.  **Note:** `Active` returns status for running jobs only, in contrast to status, which returns status for all jobs, running or not.
Parameter Keys	• *job*: Name of the job for which status should be returned (optional). • *format*:(Optional) HTML.
Accessible from JCD Home Page	Yes. Go to the JCD home page and click **"Display running jobs"**.
Example	`http://JCDServerNameorIP:JCDPort Number/active?job=wine_dgidx`

`delete`

Description	Deletes the job definition for a stopped job. After the definition is deleted, the job can no longer be restarted.
Parameter Keys	• *job*: Name of the job to delete. • *format*:(Optional) HTML.
Accessible from JCD Home Page	No.
Example	`http://JCDServerNameorIP:JCDPort Number/delete?job=wine_dgraph`

`dir`

Description	Lists the contents of a directory path.
Parameter Keys	• *path*: Path of the directory to list. • *format*:(Optional) HTML.

Accessible from JCD Home Page	Yes. Go to the JCD home page and click **"Browse the file system"**.
Example	On Windows: `http://JCDServerNameorIP:JCDPort Number/get?path=c:\endeca\platformservices\6.1.0\reference\sample_wine_data\logs\` On UNIX: `http://JCDServerNameorIP:JCDPort Number/get?path=c:\endeca\platformservices\6.1.0\reference\sample_wine_data\logs\`

`exit`

Description	Shuts down all jobs and terminates the JCD.
Parameter Keys	None.
Accessible from JCD Home Page	No.
Example	`http://JCDServerNameorIP:JCDPort Number/exit`

`get`

Description	Retrieves the complete contents of a file.
Parameter Keys	• *path*: Path of the file to return. • *offset*: (Optional) Specifies a start location different than the beginning of the file. If not specified, defaults to `0`. • *numbytes*: (Optional) Specifies the number of bytes you want to retrieve, starting at `offset`. Defaults to the entire file size.
Accessible from JCD Home Page	Yes. Go to the JCD home page and click **"Browse the file system"**.
Example	On Windows: `http://JCDServerNameorIP:JCDPort Number/get?path=c:\endeca\platformservices\6.1.0\reference\sample_wine_data\logs\wine_forge.out`

	On UNIX: `http://JCDServerNameorIP:JCDPort Number/get?path=/usr/local/endeca/current/reference/sample_wine_data/ logs/wine_forge.out`

`halt`

Description	Terminates the JCD, leaving active jobs running.  **Note:** On UNIX, If the Endeca JCD is running under init, then init will immediately restart the JCD after a halt command.
Parameter Keys	None.
Accessible from JCD Home Page	No.
Example	`http://JCDServerNameorIP:JCDPort Number/halt`

`help`

Description	Displays the help page.
Parameter Keys	None.
Accessible from JCD Home Page	Yes. Go to the JCD home page and click **"Help"**.
Example	`http://JCDServerNameorIP:JCDPort Number/help`

`printenv`

Description	Prints the environment variables that the Endeca JCD is using.
Parameter Keys	None.
Accessible from JCD Home Page	Yes. Go to the JCD home page and click **"Display environment variables"**.
Example	`http://JCDServerNameorIP:JCDPort Number/printenv`

`roll`

Description	Roll (close and re-open) the Endeca JCD log file. If you roll the log to a new file, you will also have to update your `jcd.conf` file to point to the new log location. If you don't update the `jcd.conf`, the Endeca JCD goes back to using the old log location when it is restarted.
Parameter Keys	• *path*:(Optional) New log file path (default = existing log path). • *format*:(Optional) HTML. •  **Note:** If you use the existing log path, your current log file will not be overwritten.
Accessible from JCD Home Page	No.

Example	On Windows:
	`http://JCDServerNameorIP:JCDPort Number/roll?path=c:\endeca\platformservices\workspace\logs\jcdlog.txt`
	On UNIX:
	`On UNIX: http://JCDServerNameorIP:JCDPort Number/roll?path=/usr/local/endeca/logs/jcdlog.txt`

`start`

Description	Starts a stopped job and returns the success or failure of the operation.
Parameter Keys	• *job*: Name of the job to start. • *format*:(Optional) HTML.
Accessible from JCD Home Page	Yes. Go to the JCD home page and click **"Display all defined jobs"**. Find the job you want to start and click **"Run again"**.
Example	`http://JCDServerNameorIP:JCDPort Number/start?job=wine_dgraph`

`status`

Description	Gets the status for a job. Returns the status for all jobs if no job parameter is specified.  **Note:** `Status` returns status for all jobs, running or not, in contrast to `active`, which returns status for running jobs only.

Parameter Keys	• *job*:(Optional) Name of the job for which status should be returned. • *format*:(Optional) HTML.
Accessible from JCD Home Page	Yes. Go to the JCD home page and click **"Display all defined jobs"**.
Example	`http://JCDServerNameorIP:JCDPort Number/status?job=wine_dgraph`

`stop`

Description	Stops a running job.
Parameter Keys	• *job*: Name of the job to stop. • *format*:(Optional) HTML.
Accessible from JCD Home Page	Yes. Go to the JCD home page and click either **"Display all running jobs"** or **"Display all defined jobs"**. In the **Status** line of the job in question, click **Stop**.
Example	`http://JCDServerNameorIP:JCDPort Number/stop?job=wine_dgraph`

`tail`

Description	Returns the tail (that is, the last 1 KB) of a file.
Parameter Keys	*path*: Path of the file to `tail`.
Accessible from JCD Home Page	No.

Example	On Windows:
	`http://JCDServerNameorIP:JCDPort Number/tail?path=c:\endeca\platformservices\6.1.0\reference\sample_wine_data\logs\wine_forge.err`
	On UNIX:
	`http://JCDServerNameorIP:JCDPort Number/tail?path=/usr/local/endeca/current/reference/sample_wine_data/ logs/wine_forge.err`

Related Links

Defining jobs and running the Control Interpreter on page 36

The Control Interpreter is run based on control scripts, and may require job definitions from the Endeca JCD.

The Endeca JCD home page reference on page 27

For some Endeca JCD parameters, rather than typing in parameter/value pairs, you can enter the hostname and port number (that is, `http://[host]:[port]`) in order to access the Endeca JCD home page, and then click on one of its options, explained below.

About starting jobs with the JCD

The `start` command is reported successful if a job starts, even if the job terminates due to an error.

If you use the Endeca JCD's `start` command to start a job, the JCD will start the job and report that the job has been started successfully (assuming the job was successfully started). If the job then exits, either due to normal completion or an error, that doesn't change the fact that it was started successfully by the JCD. The JCD's "successful start" report indicates only that the job started successfully. It does not indicate that the job continued to run successfully. Errors that occur after the job has been started are logged in the Endeca JCD log file.

For example, if you use the Endeca JCD `start` command to start a Dgraph job, and then the Dgraph fails while loading the index, the JCD will still report the Dgraph as successfully started, despite the fact that the Dgraph subsequently failed. The error that the Dgraph failed is logged in the Endeca JCD log file.

Related Links

Control interpreter logging on page 48

The Control Interpreter produces two types of logs to complement the Endeca JCD log (which provides information that is specific to running the Endeca JCD).

About importing SSL certificates into Internet Explorer

If you use HTTPS mode with authentication, you will need to import the SSL certificates into Internet Explorer to communicate with the JCD.

If `ssl=true` and an `sslcertfile` and `sslcafile` are specified in the `jcd.conf` file, you must import the SSL certificates into Internet Explorer in order to communicate directly with the JCD. This procedure must be followed on each Windows machine from which the Endeca JCD will be accessed.

Refer the *Endeca Security Guide* for detailed instructions on importing SSL certificates into Internet Explorer.

Note: If you are not using HTTPS mode to communicate with the Endeca JCD, you do not need to perform this procedure.

Related Links

SSL Configuration for the Control Interpreter on page 107

This appendix describes how to use SSL with the Control Interpreter.

Chapter 3

The Control Interpreter

The Control Interpreter manages the activities of multiple Endeca JCD instances, in a distributed Endeca deployment, from a single interface. This section describes how you configure and run the Control Interpreter.

Control scripts and bricks

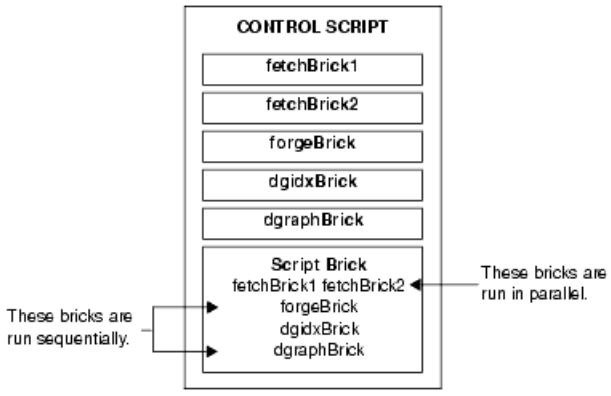
The Control Interpreter uses each machine's instance of the Endeca JCD to run jobs on that machine. Instructions for controlling all of the Endeca JCDs in a deployment reside in a control script that a developer writes for your custom environment.

A control script is made up of a collection of bricks. A brick describes a single well-defined task, such as downloading a file, indexing data, or restarting a Dgraph. Bricks translate into jobs that the Control Interpreter sends to the Endeca JCDs.

There are different bricks for different tasks: the Forge brick runs Forge, the Dgraph brick runs a Dgraph, the Shell brick runs operating system commands, and so on. Each brick has a set of attributes associated with it, most of which have default values. You can customize a brick by overriding its default attributes.

A control script always contains one or more Script bricks. Script bricks tell the Control Interpreter how to execute the other bricks in the control script file. The lines in a Script brick are executed in order. By default, bricks named on different lines are run sequentially, while bricks named on the same line are run in parallel.

In addition to bricks, a control script also contains some global default settings that are shared across bricks.



Related Links

Control Script Brick Reference on page 67

This appendix provides details about specific brick types. Some long brick settings break onto the following line in these examples; however, you should type each brick setting on a single line. If you need to wrap a line, put a space and a backslash (" \") at the break; this tells the Control Interpreter to ignore the line break.

Example control script on page 89

This simple but complete control script demonstrates how all of the Control Interpreter elements work together.

Brick names

Every brick in a control script has a name that identifies it. You use brick names to cross-reference from one brick to another, building more complex bricks out of simpler bricks.

The syntax for brick naming is:

`<brick_name> : <brick_type>`

For example, a Dgraph brick called `dgraph01` would appear as `dgraph01 : Dgraph` in the control script.

Brick names are case sensitive, and they must be unique within the control script. Names follow the same format as C identifiers (including no spaces, hyphens, or dots).

Important: If you are using multiple control scripts, brick names must be unique across your entire deployment.

Defining jobs and running the Control Interpreter

The Control Interpreter is run based on control scripts, and may require job definitions from the Endeca JCD.

Running the Control Interpreter is a one or two step process:

1. Provide the Endeca JCDs with job definitions.

   This step is not always required.
2. Run the Control Interpreter to execute jobs according to the control script logic.

Related Links

The DefineJobs utility on page 36

You must run the DefineJobs script before you run your control script for the first time, or whenever your control script changes. DefineJobs provides each instance of the Endeca JCD with the job definitions it will need in order to execute its job(s).

The DefineJobs utility

You must run the DefineJobs script before you run your control script for the first time, or whenever your control script changes. DefineJobs provides each instance of the Endeca JCD with the job definitions it will need in order to execute its job(s).

Job definitions are derived from the bricks in the control script itself.

After you have defined your jobs, you can run the control script as often as you want, without re-running DefineJobs, as long as the control script doesn't change. If you change your control script, your job definitions also change, and you must re-run DefineJobs to send the new definitions to the Endeca JCDs.

The DefineJobs utility is located in `$ENDECA_ROOT/bin` on UNIX and `%ENDECA_ROOT%\bin` on Windows.

DefineJobs syntax

The syntax for running the `DefineJobs` utility is:

`DefineJobs [options] <control_script> [command]`

where *options* represents command line options, *control_script* is the pathname to the Control Interpreter's control script, and *command* represents a specific command in the control script.

The *command* argument is optional, and may be either the name of a Script brick or a specific line within a Script brick. For example:

```
DefineJobs index.script myBrick
```

```
DefineJobs index.script dgraph.stop
```

If you do not specify a command, DefineJobs defines all of the jobs listed in the control script. If you do specify a command, then only jobs associated with that command are defined.

Related Links

DefineJobs command line options on page 37

You can change the behavior of the DefineJobs utility using the command flags described below.

DefineJobs command line options

You can change the behavior of the DefineJobs utility using the command flags described below.

By default, DefineJobs prompts you before redefining, stopping, or restarting any jobs. You use the command line options `--delete-prefix`, `--force-delete`, `--force-redefine`, `--force-restart`, and `--force-stop` to alter this behavior. In addition, you can specify control script setting overrides using the last two DefineJobs options.

Option	Description
`--job-prefix <prefix>`	Specifies a unique prefix for all JCD job names. This is useful if using the same JCD with more than one configuration file. If you want to use the `--delete-prefix` setting, you must specify prefixes with this option, rather than adding them manually.

Option	Description
`--delete-prefix <prefix>`	Delete all jobs with the given prefix. This option only works if jobs were prefixed using `--job-prefix`. Using this option means that no new jobs will be defined. Do not use this option with `--no-prompt`.
`--force-delete`	Delete unrecognized jobs without prompting.
`--force-redefine`	Do not prompt when redefining jobs.
`--force-restart`	Do not prompt when stopping and restarting jobs that are currently running but need to be redefined. This option implies `--force-stop`.
`--force-stop`	Do not prompt when stopping jobs that are currently running but need to be deleted or redefined.
`--never-delete`	Do not delete unrecognized jobs.
`--no-prompt`	Equivalent to using all three of the following: • `--force-redefine` • `--force-restart` • `--never-delete`
`--override <setting>=<value>`	Overrides a particular setting in a control script with the value specified.
`--override-file <file>`	Overrides settings in a control script with settings specified in an override file.

Related Links

About setting overrides on page 46

The Control Interpreter allows you to override brick settings on the command line. This feature is useful when you have a single control script file that you want to use in multiple environments.

About running the Control Interpreter

After the Endeca JCDs have been prepared with job definitions, you run the Control Interpreter, using the control script as an argument. The Control Interpreter instructs the individual JCDs to execute their jobs according to the control script's logic.

In Windows environments, you can run the Control Interpreter automatically from the Scheduled Tasks control panel, or manually from the command prompt. Endeca recommends that you run the Control Interpreter via the Scheduled Tasks control panel in production environments.

In UNIX environments, you run the Control Interpreter automatically from the `crontab` task scheduler, or manually from a shell prompt. Endeca recommends that you run the Control Interpreter via the `crontab` task scheduler for production environments.

The syntax for the Control Interpreter command looks like this:

`RunCommand [options] <control_script> [command]`

where *options* represents the command line options, *control_script* is the pathname of the Control Interpreter's control script file, and *command* represents the specific command in the control script to be executed.

The `RunCommand` utility is located in `$ENDECA_ROOT/bin` on UNIX and `%ENDECA_ROOT%\bin` on Windows.

The *command* argument is optional, and may be either the name of a Script brick or a specific line within a Script brick. For example:

```
RunCommand index.script myBrick
```

```
RunCommand index.script dgraph.stop
```

If you do not specify a command, RunCommand will look for a Script brick to run. If there are multiple Script bricks, RunCommand will list them, and require you to choose one. If you do specify a command, then only that command is executed.

Endeca suggests that you save the Control Interpreter output to a log file if you are running via Scheduled Tasks on Windows or via `crontab` on UNIX.

Note: In production, it is a good idea to set up your system to check the return code from RunCommand and send an email notification in case of failure.

Related Links

RunCommand utility command line options on page 39
The following table describes the command line options you can use with RunCommand:

Control interpreter logging on page 48
The Control Interpreter produces two types of logs to complement the Endeca JCD log (which provides information that is specific to running the Endeca JCD).

RunCommand utility command line options

The following table describes the command line options you can use with RunCommand:

Option	Description
`--override-file <file>`	Overrides settings in a control script with settings specified in an override file.
`--override <setting>=<value>`	Overrides a particular setting in a control script with the value specified.

Option	Description
`--debug`	A RunCommand logging option.
`--trace-line-numbers`	Prints the line number of every command as it is run, for debugging purposes.

Related Links

About setting overrides on page 46
The Control Interpreter allows you to override brick settings on the command line. This feature is useful when you have a single control script file that you want to use in multiple environments.

Control Interpreter-specific logs on page 48
By default, the Control Interpreter prints out each line of script as it runs. If you need more details, you can specify the `--debug` switch.

Communication between the Control Interpreter and JCD

The Control Interpreter initiates all communication between itself and the Endeca JCD instances. The Endeca JCD does not initiate any communication between the two.

After the Control Interpreter has sent a job to an Endeca JCD, it follows up with periodic queries to determine whether or not the job has been completed. When the JCD indicates that the job has been completed, the Control Interpreter continues with the next job in the control script.

About writing control scripts

You write control scripts using a set of standard bricks.

Developing scripts is part of the Endeca deployment process, and typically happens before the system is put into production. This section discusses common brick elements.

Related Links

Control Script Brick Reference on page 67
This appendix provides details about specific brick types. Some long brick settings break onto the following line in these examples; however, you should type each brick setting on a single line. If you need to wrap a line, put a space and a backslash (" \") at the break; this tells the Control Interpreter to ignore the line break.

Control script syntax

Control scripts are sensitive to whitespace, such as indents and carriage returns.

Indentation is significant in a control script. Lines that are more indented are considered children of lines that are less indented.

Note: To avoid issues that can occur when using certain text editors, always use tabs, not spaces, to create indentation at the beginning of lines.

Ends of lines are also significant. If you need to type a long line that will break onto the following line, put a space followed by a backslash (" \") at the point where the line breaks. This tells the Control Interpreter to ignore the line break.

Global default settings reference

The tables below describe the most common control script settings that are set globally.

You specify global default settings in the control script for settings that are shared across multiple bricks—for example, a working directory, a JCD port number, or a logging location. You can use any brick setting as a global default, as long as the setting makes sense in a global context. As necessary, you over ride a global default setting for a specific brick by specifying a different value for the setting in the brick's definition. Unless you decide to over ride a global default, the global setting applies to all bricks in a control script.

Important: You can also override global default settings and brick-specific settings from the command line.

Basic global defaults

Setting	Description
`working_machine`	The machine on which to execute the bricks. This must match the name of a Machine brick, which contains all the information necessary to connect to that machine. This setting is required for remote execution. Otherwise, if it is not specified, the bricks will run on the same machine as the Control Interpreter itself. In a distributed implementation, over ride this setting in each Machine brick. For example, in an implementation with multiple Dgraphs, you would override this setting in each Dgraph brick.
`working_dir`	The directory to use as the current directory when executing the bricks. All relative paths in the brick definitions are relative to this directory.
`stdout_base`	File prefix to indicate where `stdout` will be written. `stdout_base` may be either a full pathname and prefix, or a prefix that is relative to the *working_dir*. Each brick writes its `stdout` to a file whose name begins with this prefix, and ends with the brick name.  **Note:** You can send `stdout` and `stderr` to the same file.
`stderr_base`	File prefix to indicate where `stderr` will be written. `stderr_base` may be either a full pathname and prefix, or a prefix that is relative to the *working_dir*. Each brick writes its `stderr` to a file whose name begins with this prefix, and ends with the brick name.

Setting	Description
	Note: You can send `stdout` and `stderr` to the same file.
`append_stdout`	Specifies whether to append `stdout` or overwrite it (the default). Values are `yes` or `no`. If a brick is executing on a remote machine, this setting is ignored and `stdout` is always appended.
`append_stderr`	Specifies whether to append `stderr` or overwrite it (the default). Values are `yes` or `no`. If a brick is executing on a remote machine, this setting is ignored and `stderr` is always appended.
`endeca_bin`	Path to the `bin` subdirectory of the Endeca Platform Services distribution you are using. Setting this path is equivalent to setting the path to the `forge` binary. **Note:** This setting is required if you want to download files from remote machines via the Endeca JCD. **Note:** There may be other files, such as Perl scripts, that reside in the `bin` subdirectory along with the forge binary. In order to reference other files in the `bin` directory, you must provide a complete pathname.
`endeca_mdex_bin`	Path to the bin subdirectory of the Endeca MDEX Engine distribution you are using. Setting this path is equivalent to setting the paths to the `dgidx`, `agidx`, `dgraph`, and `agraph` binaries all at once. You can override the `endeca_mdex_bin` setting in individual bricks with the `dgidx_binary`, `agidx_binary`, `dgraph_binary`, and `agraph_binary` settings. **Note:** There may be other files that reside in the `bin` subdirectory along with the `dgidx`, `agidx`, `dgraph`, and `agraph` binaries—for example, `dgwordlist`. The `endeca_mdex_bin` setting applies only to the four core Endeca MDEX Engine binaries. In order to reference other files in the `bin` directory, you must provide a complete pathname.
`environment_vars`	Specifies the name of a Constants brick that contains the environment variables that should be used while running the system. **Note:** The environment variables are completely replaced, and not simply added to. Therefore, if you use this setting, you must specify all of the environment variables that you will need, including *PATH*.

Setting	Description
`perl_binary`	Path to the Perl interpreter to use on each machine in the deployment. This setting is required if you want to download files from remote machines via the Endeca JCD. **Note:** You must use version 5.8.3 of Perl as installed with the Endeca software. This path is used for remote file retrieval only. It is not used when running the Control Interpreter itself.
`wget_binary`	Path to the wget file-retrieval program file (installed as part of the standard Endeca installation). This setting is optional; if it is not specified in the control script, the system looks for it in the `%ENDECA_ROOT%\bin` directory on Windows and in `$ENDECA_ROOT/bin` on UNIX.
`jcd_port`	The port used to connect to the Endeca JCD. This must match the port listed in the `jcd.conf` file. The standard port for the Endeca JCD is `8088`.
`jcd_use_ssl`	Specifies whether or not the Control Interpreter must use SSL when communicating with the JCD on each machine. Values are `true` and `false`. If you have configured the JCD to use SSL (by setting `ssl=true` and specifying an `sslcertfile` in the `jcd.conf` file), then you must do the following in the control script: 1. Set `jcd_use_ssl` to true. 2. Specify an `sslcertfile`, as described below.
`sslcertfile`	Specifies the path of the certificate file, `eneCert.pem`, that Endeca components (Control Interpreter, Forge, Dgraph, and Agraph) should present when communicating with other Endeca components via SSL. You must set `sslcertfile` as a global default, although you can still override the default by setting a different `sslcertfile` within specific bricks. **Note:** In order to simplify installation and configuration, all Endeca components use the same certificate file, `eneCert.pem`, for secure communication. The `sslcertfile` you specify in `jcd.conf`, however, configures only the JCD, while the `sslcertfile` you specify in a control script dictates behavior for all other Endeca components, excluding the JCD. **Note:** This path is also used in conjunction with the `advanced forge_use_ssl` and `ene_use_ssl` settings described below.

Advanced security global defaults

The following settings are used in advanced security situations only.

Setting	Description
`forge_use_ssl`	Specifies whether Forge clients and servers should communicate with each other via SSL when running in parallel Forge mode. Values are `true` and `false`. If `forge_use_ssl` is set to `true`, you must also specify an `sslcertfile`, as described above, to indicate the location of the certificate file that Forge clients should present to Forge servers.
`ene_use_ssl`	When set to `true`, the `ene_use_ssl` setting specifies that: • The Control Interpreter should start all MDEX Engines (Dgraphs or Agraphs) with SSL flags enabled. • Clients must use SSL to communicate with MDEX Engine servers (Dgraph and Agraph). If `ene_use_ssl` is set to `true`, you must also specify an `sslcertfile`, as described above, to indicate the location of the certificate file that clients should present to MDEX Engine servers.
`sslcafile`	Specifies the path of the certificate authority file, `eneCA.pem`, that Endeca components should use to authenticate communication initiated by their clients.
`sslcipher`	Specifies the cipher the Endeca components should use when communicating with each other via SSL.

Related Links

Example control script on page 89
This simple but complete control script demonstrates how all of the Control Interpreter elements work together.

SSL Configuration for the Control Interpreter on page 107
This appendix describes how to use SSL with the Control Interpreter.

Machine brick on page 67
Machine bricks specify the name and connection details of each machine in a distributed environment.

About setting overrides on page 46
The Control Interpreter allows you to override brick settings on the command line. This feature is useful when you have a single control script file that you want to use in multiple environments.

Internal brick settings

The Control Interpreter contains internal logic that tells the Endeca JCDs how to run certain bricks.

For example, the Control Interpreter knows that the Dgraph is a continually running process, so it tells the Endeca JCDs to run a `dgraph` brick as a server process. This additional logic is internal to the Control Interpreter and does not require any special settings in the control script.

Implicit and explicit brick commands

Most bricks that are listed within a Script brick have an implicit `run` command. In order to execute this type of brick, the Script brick only has to list it in its definition.

For example:

```
fetch_and_copy_data : Script
 fetch_data_1 fetch_data_2
 copy_data_1 copy_data_2
```

Five brick types, however, have explicit commands that you use to perform operations on them: Machine, Dgraph, Archive, Agraph, and LogServer. Operations include things like starting, stopping, and testing the state of processes. In the sample Script brick below, a Dgraph brick called `dg01` is stopped and restarted.

```
restart_dg01 : Script
 dg01.stop
 dg01.start
```

Related Links

Machine brick on page 67
Machine bricks specify the name and connection details of each machine in a distributed environment.

Dgraph brick on page 77
A Dgraph brick runs the Dgraph (the MDEX Engine software).

Archive brick on page 84
The Archive brick can create, archive, and roll back directories.

Agraph brick on page 78
An Agraph brick runs the Agraph program, which defines and coordinates the activities of multiple, distributed Dgraphs.

LogServer brick on page 86
The LogServer brick controls the use of the Endeca Log Server.

Control Interpreter interaction with environment variables

The Control Interpreter automatically declares global default settings based on environment variables.

If the *ENDECA_ROOT* environment variable is set, the Control Interpreter automatically declares three global default settings: `endeca_root`, `endeca_bin`, and `dtd_dir`. Their values are as follows:

- `endeca_root` is the same as *$ENDECA_ROOT*
- `endeca_bin` is the bin subdirectory of *$ENDECA_ROOT*. This setting is used by the Forge and Fetch bricks.
- `dtd_dir` is the conf/dtd subdirectory of *$ENDECA_ROOT*. This setting is used by Forge bricks.

If the *ENDECA_MDEX_ROOT* environment variable is set, the Control Interpreter automatically declares the `endeca_mdex_root` and `endeca_mdex_bin` global default settings. Their values are as follows:

- `endeca_mdex_root` is the same as *$ENDECA_ROOT*
- `endeca_mdex_bin` is the bin subdirectory of *$ENDECA_MDEX_ROOT*. This setting is used by the following bricks: Dgidx, Dgraph, Agidx, Agraph, and AgraphIndex.

If a script specifies its own value for any of the above settings, that value takes precedence over the *$ENDECA_ROOT* or *$ENDECA_MDEX_ROOT* setting.

About setting overrides

The Control Interpreter allows you to override brick settings on the command line. This feature is useful when you have a single control script file that you want to use in multiple environments.

In general, override settings are specified only for items that vary among multiple environments. Endeca recommends that, if a setting has different values in different environments, you omit a value for the setting in the control script. Instead, the value is be provided by the override settings.

When you specify overrides, you can provide either the new value for a specific setting on the command line itself, or the name of a file that contains setting overrides.

 **Note:** In most circumstances, overrides are provided using an override file.

The syntax for specifying a file containing overrides is:

`--override-file <pathname>`

where <*pathname*> is the full pathname to the override file.

The syntax for overriding a specific setting is:

`--override <setting>=<value>`

where <*setting*> is the setting you want to override and <*value*> is the new value you want the Control Interpreter to use.

You can use `--override` and `--override-file` with both the DefineJobs utility and the RunCommand utility.

About specifying settings in an override file

You can specify both brick-specific and global default setting overrides within an override file.

The syntax for a brick-specific override is:

`<brickName>.<setting>=<value>`

where <*brickName*> is the name of the brick whose setting will be overridden, <*setting*> is the specific setting to override, and <*value*> is the value you want the Control Interpreter to use.

The syntax for a global default setting in an override file is:

`<setting>=<value>`

where <*setting*> is the global default setting to be overridden, and <*value*> is the value you want the Control Interpreter to use instead.

 **Note:** Setting overrides can be empty.

About setting priority

The Control Interpreter checks several locations for a brick's settings.

The Control Interpreter logic for determining the value of a brick setting follows this path:

1. Get the value from a brick-specific override.
2. If no brick-specific override exists, get the value from the brick definition in the control script.

3. If the setting does not exist in the brick's control script definition, get the value from the global default specified in the override file.
4. If no global default is specified in the override file, get the value from the global default settings specified in the control script.

About handling repetition in control scripts

At times it is necessary to include repetition in your control scripts. By using specific syntax, you can employ text substitution to avoid typing out nearly-identical lines in your scripts.

For example, you might want to do the same thing on ten machines. Rather than type out ten separate brick definitions, you can automatically repeat certain lines of the control script, with text substitution for each repetition.

The syntax used to handle repetition in control scripts is as follows:

```
$(replace PATTERN with foo bar quux)
<lines to expand>
```

PATTERN is a text string that will be replaced wherever it occurs within the lines to be expanded. Any pattern can be chosen, as long as it is a valid identifier using only letters and numbers. `with` is a keyword that separates the pattern from the list of replacements. "`foo bar quux`" is a list of replacements.

For example, the following lines:

```
$(replace PATTERN with hello world)
print "PATTERN";
```

would expand into this:

```
print "hello";
print "world";
```

In this example, the control script archives the "logs" directory on several Windows machines.

```
archive_logs : Script
 parallel
  $(replace MACHINE with idx01 idx02 idx03)
   archive_logs_MACHINE

$(replace MACHINE with idx01 idx02 idx03)
 archive_logs_MACHINE : Shell
  working_machine = MACHINE
  move logs logs.old
  mkdir logs
```

Variable references in repetition syntax

Instead of typing out an explicit list of replacement terms, you can use a variable reference.

For example:

```
machines = idx01 idx02 idx03 $(replace MACHINE with $(machines)) ...
```

Important: If you put the machines setting in a Constants brick, then the Constants brick must be declared before the replace statement or it will not work. Only the `replace syntax` requires

variables to be declared before they are used—for all other control script features, order is not important.

Control interpreter logging

The Control Interpreter produces two types of logs to complement the Endeca JCD log (which provides information that is specific to running the Endeca JCD).

The Control Interpreter produces two types of logs:

- Logs that are specific to the Control Interpreter software itself.
- Logs that provide detailed information about each job that is run.

Related Links

About logging and monitoring on page 20

The Endeca JCD records informational, warning, and error messages about its operations in a JCD log. Each instance of the Endeca JCD records its logs on its local machine.

Control Interpreter-specific logs

By default, the Control Interpreter prints out each line of script as it runs. If you need more details, you can specify the `--debug` switch.

The switch directs the Control Interpreter to print explicit information about its actions, including start and stop times and the success or failure of processes.

`RunCommand --debug <control_script> [command]`

The default Control Interpreter logging information is printed to `stdout`. The information from `--debug` is printed to `stderr`.

The Control Interpreter log information prints to the screen, by default. However, you can redirect `stdout` or `stderr` to a file by using `>stdout.log` and `2>stderr.log` on the command line. Endeca recommends that you redirect `stderr` if you use `--debug` because the information that `--debug` yields can be very long.

Job-specific logs

Job-specific logs provide detailed information about each job that is run via the Control Interpreter.

For example, the log for a Dgidx job contains detailed information about the dimensions and properties that were incorporated into the MDEX Engine indices during indexing.

You specify where a job's log will be stored in either the default settings, using `stdout_base` and `stderr_base`, or in the individual brick definitions, using `stdout` and `stderr`. A log file is written out for each job on the local machine that executed the job.

Chapter 4

Running Implementations with a Control Script

This section documents how to run an Endeca application using a Control Interpreter script. Note that in these instructions, UNIX commands that are longer than a single line are broken with a backslash character (\) with no surrounding whitespace. You do not need to type the backslash character; however, if you do, it will not affect the command.

Overview of running Endeca components

Running the Endeca components is a three-step process. In the reference implementation, the three programs are launched by a single Control Interpreter script, using the Endeca JCD service.

For the sake of simplicity, this procedure assumes that you have installed the Endeca MDEX Engine and Endeca Platform Services on the same machine.

To run the Endeca components:

1. Run the Forge program to standardize and model the raw data.
2. Run the Dgidx program to index the data.
3. Run the Dgraph program to start the Endeca MDEX Engine.

Running Endeca components with a single control script

You can run Forge, Dgidx, and Dgraph by using a single control script.

For the sake of simplicity, this procedure assumes that you have installed the Endeca MDEX Engine and Endeca Platform Services on the same machine.

On UNIX, the Endeca reference implementations are installed in the `$ENDECA_REFERENCE_DIR` directory (`%ENDECA_REFERENCE_DIR%` on Windows). This section uses the `sample_wine_data` reference implementation. It assumes that the full path of the Control Interpreter script is as follows:

Windows:

```
%ENDECA_REFERENCE_DIR%\sample_wine_data\etc\remote_index.script
```

UNIX:

```
$ENDECA_REFERENCE_DIR/sample_wine_data/etc/remote_index.script
```

To run an Endeca implementation with a control script:

1. Set the *ENDECA_MDEX_ROOT* environment variable.
   - On Windows, make sure you created an *ENDECA_MDEX_ROOT* system environment variable and restarted the JCD Service.
   - On UNIX, make sure that you have run the MDEX Engine script that sets the *ENDECA_MDEX_ROOT* environment variable.
2. Edit the `remote_index.script` control script:
   a) Make sure that `jcd_use_ssl` is set to `false`.
   b) If the Endeca JCD's default port of 8088 (which appears in the `jcd_port=8088` line of the Global Variables section) is already in use, change it to an unused port number.

      Make sure it matches the port setting in the corresponding `jcd.conf` file (located in the *ENDECA_ROOT*`/workspace/etc` directory).
   c) If the Dgraph's default port of 8000 (which appears in the `dgraph_port=8000` line of the Global Variables section) is already in use, change it to an unused port number.
   d) If the Log Server default port of 8002 (which appears in the `logserver_port=8002` line of the Global Variables section) is already in use, change it to an unused port number.

      The Log Server's port must be two greater than the Dgraph's port. For example, if you modify the Dgraph's port to 9090, then the Log Server's port must be 9092.
   e) Save your changes and close the script.
3. In a command prompt, run the `mdex_setup` script to export the MDEX Engine environment variables.

   The script is located in the MDEX Engine's root directory. This ensures that the command prompt has the correct MDEX Engine environment settings.
4. Run the DefineJobs script with the path of the `remote_index.script` control script as its parameter:

Option	Description
Windows:	`DefineJobs %ENDECA_REFERENCE_DIR%\sample_wine_data\etc\remote_index.script`
UNIX:	`DefineJobs $ENDECA_REFERENCE_DIR/sample_wine_data/etc/remote_index.script`

   The DefineJobs script provides each instance of the Endeca JCD with the job definitions it will need in order to execute its job(s). Job definitions are derived from the bricks in the control script itself.

   After a successful run, you should see something similar to the following output:

```
Getting job definitions from JCD on host localhost...
Created job wine_dgidx on host localhost
Created job wine_dgidx.aspell on host localhost
Created job wine_dgidx.copy_aspell_data_files on host localhost
Created job wine_dgraph on host localhost
Created job wine_fetch on host localhost
Created job wine_forge on host localhost
Created job wine_genreport on host localhost
Created job wine_logserver on host localhost
Created job wine_toolsreport on host localhost
All job definitions are correct.
```

5. Run the RunCommand script on the `remote_index.script` control script to start the `runme` script brick:

Option	Description
Windows:	`RunCommand %ENDECA_REFERENCE_DIR%\sample_wine_data\etc\remote_index.script runme`
UNIX:	`RunCommand $ENDECA_REFERENCE_DIR/sample_wine_data/etcremote_index.script runme`

   This command launches the Control Interpreter and instructs it to execute the jobs defined in the previous step. This runs Forge, Dgidx, Dgraph, and Log Server programs to process the sample data and starts an Endeca MDEX Engine.

   After a successful run, you should see something similar to the following output:

```
Checking that required jcd jobs are defined correctly...
[Sep 16 10:36:42] runme
[Sep 16 10:36:42]   wine_forge
[Sep 16 10:37:06]   wine_dgidx
[Sep 16 10:38:53]   if wine_dgraph.running
[Sep 16 10:38:55]   wine_fetch
[Sep 16 10:38:58]   wine_dgraph.start
[Sep 16 10:39:02]   if wine_logserver.running
[Sep 16 10:39:04]   wine_logserver.start
Script completed successfully.
```

You can verify that your installation is running correctly by using the JSP reference implementation shipped with Platform Services. To use this reference, make sure that Endeca HTTP Service is running and then use a browser with a URL similar to this example:

```
localhost:8888/endeca_jspref
```

Related Links

Setting the ENDECA_MDEX_ROOT environment variable on page 18

If you have installed the Endeca Control System on a Windows machine, it is recommended that you set *ENDECA_MDEX_ROOT* as a system environment variable, so that the JCD Service uses it when you start the service.

Generating a log report

This section describes how to run the Log Server and Report Generator using the Endeca reference implementation.

To run the Log Server and Report Generator:

1. Launch an Endeca reference implementation in a Web browser.
2. Generate logging information by browsing the sample wine data. Perform several queries against the sample wine data by clicking **dimensions**, **properties**, **merchandising features**, or other options.
3. From a command prompt, type the following command to run the Control Interpreter on the `re¬ mote_index.script` control script and start the `wine_genreport` brick:

Option	Description
Windows:	`RunCommand %ENDECA_REFERENCE_DIR%\sample_wine_data\etc\remote_index.script wine_genreport`
UNIX:	`RunCommand $ENDECA_REFERENCE_DIR/sample_wine_data/etc/remote_index.script wine_genreport`

This launches the Report Generator program to process the sample data log files.

After a successful run, you should see something similar to the following output:

```
Checking that required jcd jobs are defined correctly...
[Sep 16 10:50:36] wine_genreport
Script completed successfully.
```

4. Open the `sample_report.html` file to view the log report.
   - In Windows, the file is located in the `%ENDECA_REFERENCE_DIR%\sample_wine_data\reports` directory.
   - In UNIX, the file is located in the `$ENDECA_REFERENCE_DIR/sample_wine_data/reports` directory.

Chapter 5

Configuring and Viewing Reports in a Control System Environment

This chapter describes how to configure and run the Report Generator in an Endeca Control System environment.

Overview of logging and reporting

The architectural concepts, API usage, and report configuration of the Endeca Logging and Reporting system are largely the same in both an Endeca Application Controller environment and a Control System environment.

This section describes the specific differences of logging and reporting in a Control System environment.

For general information about implementing logging and reporting, see the *Endeca Log Server and Report Generation Guide*.

About configuring and running the Log Server

The Log Server can be configured through the LogServer brick.

The Log Server can be run from the JCD, if the LogServer brick has `working_machine` set. If you set the Log Server up in this way, the JCD will restart the Log Server automatically if it crashes.

As soon as the Log Server starts, it attempts to open a log file and write a header and timestamp. If this fails, it exits immediately, without accepting any requests. The Log Server begins a new file when you issue the `roll` command, or automatically if the current file becomes larger than 1 GB.

The log file name is a combination of the current date and time and the log file prefix that you specify, in the format `prefix.timestamp`. The timestamp indicates when the particular file was started and makes it possible to distinguish among multiple log files.

The Log Server requires no configuration file. To start the Log Server, you give it a port and a file prefix through the LogServer brick or on the command line.

There is no default logging directory, although typically you would create one as part of your application development.

Related Links

Working With the Endeca JCD on page 17
In a UNIX environment, the Endeca Job Control Daemon (JCD) monitors and manages the Endeca Information Access Platform software to provide a robust process execution environment. In a Windows environment, the analogous component is implemented as the Endeca JCD Service. This section discusses the administrator's interaction with the Endeca JCD.

About running the Log Server from the command line on page 54
At times you might want to communicate directly with the Log Server through the command line. If you choose to do this, keep in mind that the JCD cannot automatically restart the Log Server if it is started at the command line.

About running the Log Server from control scripts on page 54
The Log Server is can be started and managed by the Control Interpreter's LogServer brick.

About running the Log Server

You can run the Log Server in two ways: through Control Interpreter scripts, using the LogServer brick, or through the command line. The former is recommended.

About running the Log Server from control scripts

The Log Server is can be started and managed by the Control Interpreter's LogServer brick.

Using the LogServer brick, you can specify:

- The port to which the Log Server should listen for log requests.
- The file path prefix for any log files output by the Log Server.
- Whether you want the log files compressed by the gzip utility.

The LogServer brick also has a set of commands that allow you to:

- Start and stop the Log Server.
- Determine if the Log Server is running.
- Roll the logs created by the Log Server.

Related Links

LogServer brick on page 86
The LogServer brick controls the use of the Endeca Log Server.

About running the Log Server from the command line

At times you might want to communicate directly with the Log Server through the command line. If you choose to do this, keep in mind that the JCD cannot automatically restart the Log Server if it is started at the command line.

The Log Server executable is `$ENDECA_ROOT/bin/logserver` on UNIX and `%ENDECA_ROOT%\bin\logserver.exe` on Windows.

The command for running the Log Server is:

```
logserver --port <port> --log-file-prefix <fileprefix>
```

The two arguments, which are required, are:

- `--port <port>` is the port to which the Log Server listens for requests. It must be a port number less than or equal to 32767.
- `--log-file-prefix <fileprefix>` is the file path prefix to use for log files.

In addition to the two required arguments, there is an optional command line argument. The `--gzip` argument compresses the generated log files using gzip compression and adds a `.gz` suffix to the log file name.

Important: If you use the `--gzip` option, logs will not be written to disk as soon as they are received. Therefore, if the Log Server crashes unexpectedly, you may lose some log entries.

About monitoring the Log Server

You can check if the Log Server is running from the address bar.

To check that the Log Server is running, issue the following URL:

`http://LogServerNameorIP:LogServerPortNumber/stats`

If the Log Server is running, this URL returns a confirmation message containing the file name, number of log entries, and number of errors. If it is not running, you will see your browser's default error message.

About rolling the Log Server

You can roll the Log Server from the address bar by appending the `/roll` command.

To roll the Log Server, issue the following URL:

`http://LogServerNameorIP:LogServerPortNumber/roll`

Configuring report contents and format

You can customize the content of a report in either an Endeca Application Controller environment or in a Control System environment.

For information on the EAC, see the *Endeca Application Controller Guide*.

About generating reports

There are two ways to generate reports in a Control System environment.

You can run the Report Generator using these methods:

- Using the ReportGenerator brick in a Control Interpreter script.
- Manually from a command prompt on Windows or a shell prompt on UNIX.

Related Links

ReportGenerator brick on page 88

The ReportGenerator brick runs the Report Generator, which processes Log Server files into HTML-based reports that you can view in your Web browser and XML reports that you can view in Endeca Workbench.

Automating report generation

You can automate report generation just as you would automate any other task on your operating system.

If you use either the Control Interpreter or the command line to generate reports, you may want to automate the process using the **Scheduled Tasks** control panel on Windows or `crontab` task scheduler on UNIX. See your operating system documentation for details about automated scheduling.

Generating reports from control scripts

You can run the Report Generator in a Control Interpreter control script using the ReportGenerator brick. This section describes how to run an existing ReportGenerator brick.

The `sample_wine_data` control script, `remote_index.script`, includes a `wine_genreport` brick, which generates an HTML report using the `report_stylesheet.xsl` stylesheet.

To generate records from the ReportGenerator brick:

1. Use the following command to run DefineJobs on the wine reference implementation:

Option	Description
Windows:	`DefineJobs /%ENDECA_REFERENCE_DIR%\sample_wine_data\etc\remote_index.script`
UNIX:	`DefineJobs ENDECA_REFERENCE_DIR/reference/sample_wine_data/etc remote_index.script`

   Job definitions, including running the Report Generator, are derived from the control script.

   A successful run produces output similar to the following:

   ```
   Getting job definitions from JCD on host localhost...
   Created job wine_dgidx on host localhost
   Created job wine_dgidx.aspell on host localhost
   Created job wine_dgidx.copy_aspell_data_files on host localhost
   Created job wine_dgraph on host localhost
   Created job wine_fetch on host localhost
   Created job wine_forge on host localhost
   Created job wine_genreport on host localhost
   Created job wine_logserver on host localhost
   All job definitions are correct.
   ```

2. Use the following command to run the wine_genreport brick on the wine reference implementation and generate an HTML report:

Option	Description
Windows:	`RunCommand %ENDECA_REFERENCE_DIR%\sample_wine_data\etc\remote_index.scri wine_genreport`
UNIX:	`RunCommand ENDECA_REFERENCE_DIR/sample_wine_data/etc/remote_index.script/wine_genrep`

After running the command, a file named `sample_report.html` exists in the `%ENDECA_REFERENCE_DIR%\sample_wine_data\reports` on Windows or `$ENDECA_REFERENCE_DIR/sample_wine_data/reports` on UNIX. You can open this file in any Web browser.

Related Links

ReportGenerator brick on page 88

The ReportGenerator brick runs the Report Generator, which processes Log Server files into HTML-based reports that you can view in your Web browser and XML reports that you can view in Endeca Workbench.

Report Generator command line options

You can run the Report Generator manually from the command prompt on Windows or a shell prompt on UNIX.

There is a `.bat` file for Windows or a `.sh` file on UNIX installed in `%ENDECA_ROOT%\bin` that you use to run the Report Generator. The syntax for the Report Generator utility is as follows:

`ReportGenerator [options]`

[options] represents additional command-line options you use to control report generation. The following tables describe these options, some of which are required and some of which are optional.

Required settings

Option	Description
`--logs <logpath>`	Path to the input log file. If this is a directory, then all log files in that directory are read.
`--output <filepath>`	Complete path, including the filename of where to store the generated report.
`--stylesheet <filepath>`	Complete path, including the filename to the file that specifies how to format the generated report.

Optional settings

Option	Description
`--settings <settings file>`	Complete path to `report_settings.xml`, including the filename. This file specifies which report sections and items to exclude, if any. See the *Endeca Log Server and Report Generation Guide* for more information. If unspecified, the Report Generator creates a report with the following defaults: • It includes all report sections and items. • Top `N` values are 10, 20, 50, and 100.

Option	Description
	• The default session queue size is set to 5000.
`--timerange <keyword>`	Set the time span of interest (or report window). Allowed keywords: • `yesterday` • `last-week` • `last-month` • `day-so-far` • `week-so-far` • `month-so-far` These keywords assume that days end at midnight, and weeks end on the midnight between Saturday and Sunday.
`--start-date <date>` `--stop-date <date>`	These set the report window to the given date and time. The date format should be either `yyyy_mm_dd` or `yyyy_mm_dd.hh_mm_ss`. For example, 2007_04_23.19_30_57 expresses April 23, 2007 at 7:30:57 in the evening. The `--stop-date` parameter is exclusive. This means that that if you only want to report a single date (e.g., 2/17/2009), you have to specify the `--start-date` as 2/17 and the `--stop-date` as 2/18.
`--time-series <frequency>`	Generates time series data at a specified frequency. The value can be either `hourly` or `daily`.
`--charts <status>`	Generates charts in reports. The value is either `enable` or `disable`. If unspecified, the default is set to `disable`.

About displaying reports

To display an HTML report, open it in any Web browser.

Related Links

About generating reports for Endeca Workbench on page 58
Although it is not recommended, you can generate XML-based reports in a Control System environment.

About generating reports for Endeca Workbench

Although it is not recommended, you can generate XML-based reports in a Control System environment.

Endeca recommends that if you want to display XML reports in Endeca Workbench, you use the Endeca Workbench to run the Log Server and Report Generator, as described in the *Endeca Log Server and Report Generation Guide*.

Note: You must run both the Log Server and the Report Generator from the same environment. In other words, you cannot run the Log Server from a control script and then run the Report Generator from Endeca Workbench, or vice-versa.

About generating reports in XML

The process for generating XML reports is similar to generating other report types, with two differences.

In order to generate XML reports, you must:

- Specify `tools_report_stylesheet.xsl` as your stylesheet.
- Follow specific filename and location requirements.

If you are running reports from the command line, point to `tools_report_stylesheet.xsl` when you enter the command. If you are running reports from a control script, you will have to edit the ReportGenerator brick to use `tools_report_stylesheet.xsl`.

The filename and location requirements are:

- Daily and weekly report files must be output to `\workspace\reports\daily` and `\workspace\reports\weekly` subdirectories, respectively.
- The file name of the reports in those directories must have a timestamp format of `yyyymmdd.xml`.

This Windows example shows the directory structure and file naming of a reports directory that contains both daily and weekly reports intended for view in Endeca Workbench:

```
workspace\reports\daily\20040701.xml
.....................\20100702.xml
.....................\20100703.xml

workspace\reports\weekly\20100704.xml
..................... \20100711.xml
..................... \20100718.xml
```

About viewing reports in Endeca Workbench

As long as you follow the filename and location requirements, you will be able to see your reports in Endeca Workbench.

For information on viewing reports in Endeca Workbench, see the *Endeca Workbench Help*.

Chapter 6

Common System Architectures in an Endeca Implementation

This chapter describes typical system architectures for each stage of an Endeca implementation.

Overview of system architectures

This section provides a general description of typical system architectures for each stage of an Endeca implementation.

Endeca implementations typically have three stages:

1. Development
2. Staging and testing
3. Production

This section does not provide specific system sizing requirements for a particular implementation. There are too many variables in each unique implementation to give general guidance. Some of these variables include hardware cost restrictions, data processing demands, application throughput demands, query load demands, scale requirements, failover availability, and so on. Endeca Professional Services can perform a hardware sizing analysis for your implementation.

Development environment

A development environment is one in which developers create or substantially modify an Endeca implementation.

This implementation does not serve end-user queries. Because data processing and query processing demands are not very important at this stage, development typically occurs on a single machine. The single machine runs the JCD (deprecated), Forge, Dgidx, the Endeca Application Controller, a Web server, and the MDEX Engine.

Staging and testing environment

A staging environment is one that validates the correctness of the implementation including data processing and all necessary search and navigation features.

Features such as merchandising, thesaurus entries, and others may require business users to modify the implementation during this implementation phase. This environment is also typically used to test performance of the system. Once the implementation works as required, it is migrated to the production environment.

In terms of hardware architecture, most staging environments closely resemble or exactly match the intended production environment. This means the production environment typically determines the architecture of the staging environment.

Sample production environments

A production environment is a live Endeca implementation that serves end-user search and navigation queries.

There are a variety of system architectures in a production environment. All of them typically use at least two servers and one load balancer. As system demand increases, the number of servers necessary in the implementation increases. Demand may take the form of time to crawl source data, frequent source data updates, faster query throughput, faster response time under increasing load, and so on. Several of the most common implementation architectures are described in the following sections.

Descriptions of implementation size

We can roughly divide implementations into small, medium, and large.

A full definition of these terms includes an accounting of record size (number and size of properties and dimension values per record), total data set size, the number of indexing and MDEX Engine servers, and other measurements of scale.

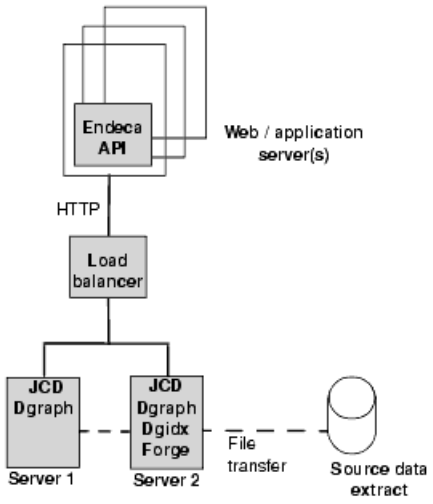
Although that level of detail is necessary for sizing a specific implementation, it is not necessary for the more general discussion of hardware architecture here. For simplicity's sake, this chapter uses the terms small, medium, and large as follows:

- A small implementation means the Dgraph runs an application's data set on a single processor.
- A medium implementation means a single Dgraph is mirrored several times for throughput (rather than solely for redundancy), and it means a dedicated server may be necessary for crawling or indexing.
- A large implementation means a data set must be partitioned into multiple Dgraphs (that is, an Agraph implementation) and a dedicated machine is required for crawling or indexing.

Small implementation with lower throughput

A simple architecture for smaller implementations is made up of two servers and a single load balancer.

Server 1 runs only the MDEX Engine. Server 2 runs a mirror of the MDEX Engine (for redundancy) and Forge and Dgidx. A single load balancer distributes queries between the MDEX Engines on servers 1 and 2.



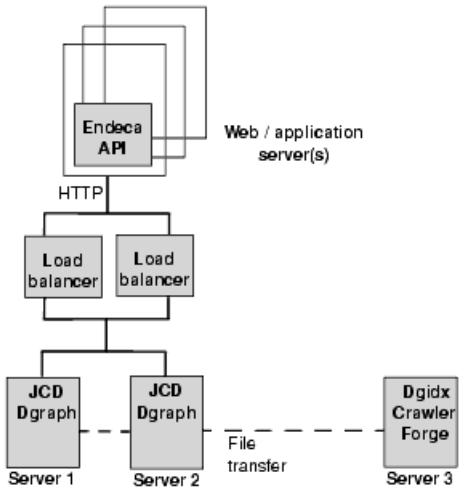
The advantage of this scenario is low cost and MDEX Engine redundancy. If one MDEX Engine is offline for any reason, the load balancer distributes user queries to the other MDEX Engine.

The disadvantage of this scenario is that the system operates at reduced throughput capacity during Forge and Dgidx processing, and during a server failure of either machine. Also, if the single load balancer fails, the system goes offline.

Small implementation using a crawler

In this example system architecture, a small implementation is made up of three servers and two load balancers.

Servers 1 and 2 run mirror copies of the MDEX Engine. Two load balancers distribute incoming user queries to the MDEX Engines. If either load balancer or MDEX Engine should fail, then the redundant load balancer or MDEX Engine handles all queries. Server 3 runs all the offline processes including the crawler, Forge, and Dgidx.



There are several advantages of this scenario. First, the MDEX Engine is mirrored, and each MDEX Engine runs on a dedicated server, so the servers do not need to share resources for Forge processing

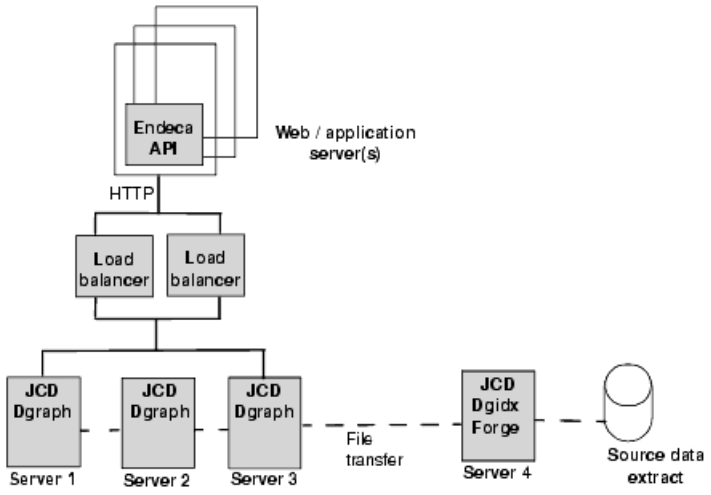
and indexing. Second, this scenario employs two load balancers to reduce potential offline time if one balancer fails. Lastly the processes to crawl source data, run Forge, and run Dgidx occurs on a single server that is not shared by a MDEX Engine.

The disadvantage of this scenario is that the implementation operates at reduced throughput if either MDEX Engine server fails.

Medium implementation with higher throughput

In this example system architecture, a medium implementation that requires higher query throughput is made up of four servers and two load balancers.

To achieve higher throughput, servers 1, 2, and 3 all run mirror copies of the MDEX Engine. This level of redundancy provides faster throughput by load balancing the incoming queries over a greater number of MDEX Engines. If either load balancer or any MDEX Engine should fail, then the redundant load balancer and remaining MDEX Engines handle all queries. Server 4 runs all the offline processes including Forge and Dgidix.



The advantage of this scenario is that overall throughput and redundancy is high. Each MDEX Engine runs on a dedicated server, so the servers do not need to share resources for Forge processing and indexing. Also, this scenario employs two load balancers to reduce potential offline time if one balancer fails.

The disadvantage of this scenario is that the implementation operates at reduced throughput if any MDEX Engine server fails. However, a single server failure has less effect on the implementation than the previous examples because the MDEX Engine has been replicated more times than in previous examples.

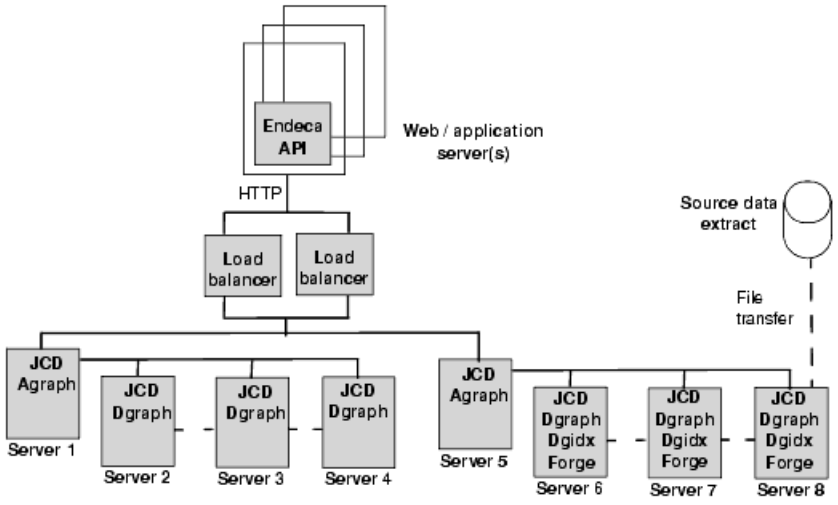
Large implementation using an Agraph

In this example system architecture, a large implementation requires a data set that is partitioned over several MDEX Engine servers and controlled by an Aggregated MDEX Engine (Agraph). The implementation is made up of eight servers and two load balancers.

The eight servers are grouped into two clusters of four servers per cluster. Each cluster has three servers running a partition of the total data set. The remaining server in each cluster runs an Agraph

to coordinate their respective cluster partitions. Each cluster mirrors the other's MDEX Engines; however, one of the clusters also runs the Forge and Dgidx processes.

Two load balancers distribute queries to both clusters. If either load balancer fails, then the redundant load balancer distributes all queries.



There are several advantages of this scenario. First, the cluster of MDEX Engines is redundant and one cluster of MDEX Engines runs on dedicated servers, so those servers do not need to share resources for Forge processing and indexing. Second, this scenario employs two load balancers to reduce potential offline time if one balancer fails.

The disadvantage of this scenario is that the system operates at reduced throughput during Forge and Dgidx processing. Also, if one MDEX Engine in a cluster fails, that entire cluster goes offline, and the system operates at reduced capacity while the remaining cluster services all queries.

Appendix A

Control Script Brick Reference

This appendix provides details about specific brick types. Some long brick settings break onto the following line in these examples; however, you should type each brick setting on a single line. If you need to wrap a line, put a space and a backslash (" \") at the break; this tells the Control Interpreter to ignore the line break.

Machine brick

Machine bricks specify the name and connection details of each machine in a distributed environment.

If you are connecting to multiple machines, you usually set the `jcd_port`, `jcd_use_ssl`, and `sslcertfiles` settings globally, because they tend to be the same across machines. However, if machines with different JCD configurations need to communicate, these settings may be specified in individual Machine bricks. An example of such a configuration would be one machine running on port 8088 with certificate files located on its C: drive, a second running on port 9099 with certificate files located on its D: drive, and a third running on port 7077 without SSL.

Machine brick settings

Setting	Description
`name`	IP address or DNS name of the machine. This setting is optional, and defaults to the brick name. If you choose to omit this setting, the brick name must be the same as the machine's DNS name.
`jcd_port`	The port used to connect to the Endeca JCD. This must match the port listed in the `jcd.conf` file. The standard port for the Endeca JCD is 8088.  **Note:** Although `jcd_port` is usually set globally, you may set it individually for a specific Machine brick.
`jcd_use_ssl`	Specifies whether or not the Control Interpreter must use SSL when communicating with the JCD on this machine. Values are `true` and `false`.

Setting	Description
	If you have configured the JCD to use SSL (by setting `ssl=true` and specifying an `sslcertfile` in the `jcd.conf` file), then you must do the following in the control script: 1. Set `jcd_use_ssl` to `true`. 2. Specify an `sslcertfile`, as described below.  **Note:** Although `jcd_use_ssl` is usually set globally, you may set it individually for a specific Machine brick.
`sslcertfile`	Specifies the path of the certificate file, `eneCert.pem`, that Endeca components (Control Interpreter, Forge, Dgraph, and Agraph) should present when communicating with other Endeca components via SSL. There must be a global default set for `sslcertfile`, but you can override the default by setting a different `sslcertfile` within specific bricks.  **Note:** In order to simplify installation and configuration, all Endeca components use the same certificate file, `eneCert.pem`, for secure communication. The `sslcertfile` you specify in `jcd.conf`, however, configures only the JCD, while the `sslcertfile` you specify in a control script dictates behavior for all other Endeca components, excluding the JCD.  **Note:** This path is also used in conjunction with the advanced `forge_use_ssl` and `ene_use_ssl` settings.
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

Machine brick commands

Machine bricks have three commands that can be used within Script bricks:

Command	Description
`is_win32`	The `is_win32` operation succeeds if the machine that the Machine brick is defined for is running the Windows operating system.  **Note:** This operation allows you to write control scripts that are compatible with both the Windows and UNIX platforms. See the examples below.

Command	Description
`is_unix`	The `is_unix` operation succeeds if the machine that the Machine brick is defined for is running a UNIX operating system.  **Note:** This operation allows you to write control scripts that are compatible with both the Windows and UNIX platforms. Seethe examples below.
`halt_jcd`	The `halt_jcd` operation stops the Endeca JCD on the machine for which the Machine brick is defined. On Windows, the Endeca JCD will be restarted automatically by the Windows Service Manager after the `halt_jcd` command is executed. On UNIX, the Endeca JCD is automatically restarted via the inittab.  **Note:** The `halt_jcd` command is an advanced feature that is generally used for automated software updating only.

The following is an example of a Machine brick called indexer:

```
indexer : Machine
 name = idx01 #DNS name of machine
 jcd_port = 8088
```

The following example shows what a Machine brick might look like if the jcd_port setting was specified globally:

```
idx01 : Machine
 name = idx01 #DNS name of machine
```

The following excerpt from a Script brick illustrates how Machine operations work. This example shows how to test for a specific platform, and control what the Control Interpreter does depending on the results of the test.

```
myScript : Script
 if idx01.is_win32
  do_win32_version
 else
  do_unix_version
```

Related Links

Global default settings reference on page 41
The tables below describe the most common control script settings that are set globally.

Fetch brick

A Fetch brick is used to retrieve raw data for processing. You must use a separate Fetch brick for each raw data source.

Fetch brick settings

Setting	Description
source	A URL specifying where and how to retrieve the data. Protocols understood are file, HTTP, and FTP. Secure protocols understood are files and HTTPS. The file and files protocols either move or copy files, depending on the value of remove_source. File protocol paths can be relative (for example, file:///foo) or absolute (for example, file:////foo). The file and files protocols support file retrieval from remote machines via the Endeca JCD (see "Fetching files from remote machines" below). For the file, files, and FTP protocols, if the source contains wildcards like "*" or "?", then all files matching that pattern are retrieved. In this case, the dest setting must name a directory.  **Note:** Due to the nature of their content, HTTP and HTTPS URLs cannot use wildcards.
username	Required for FTP URLs. Optional for HTTP and HTTPS URLs. Not used for file and files URLs.
password	Required for FTP URLs. Optional for HTTP and HTTPS URLs. Not used for file and files URLs.
dest	The file or directory in which the fetched files should be stored. If dest names a directory, the directory must already exist. The dest setting for file, files, and FTP URLs that use wildcards must be a directory. For any URL that does not use wildcards, the dest setting must be equivalent to the source; in other words, if the source specifies a directory, the dest must also be a directory. If the source specifies a file, the dest must also be a file.
remove_source	Specifying this Boolean setting deletes the source after it has been fully and successfully downloaded. remove_source is only supported for removing files from the local machine. It is not supported for HTTP, HTTPS, or FTP URLs.
recursive	If this Boolean setting is specified, directories are downloaded recursively. This setting is only supported for file and files URLs, both local and remote.
stdout	Where to redirect stdout for the brick. By default, stdout is sent to the screen. Specifying a value for stdout overrides the stdout_base setting.
stderr	Where to redirect stderr for the brick. By default, stderr is sent to the screen. Specifying a value for stderr overrides the stderr_base setting.

Fetching files from remote machines

Fetch bricks support file retrieval from remote machines via the Endeca JCD. In order to use this functionality, however, you must set certain brick settings correctly:

Fetch brick settings:

- `source` must specify a remote file URL. If the Endeca JCD on the remote machine is configured to use SSL, you must use the `files` protocol.
- `dest` must specify either a filename (for retrieving a single file) or a directory name (if you are using wildcards in your source setting).

Default settings:

- `endeca_bin` must be set to the pathname of the `bin` directory in the Endeca Platform Services installation you are using.
- `jcd_port` must be set to the Endeca JCD port on the remote machine.
- `perl_binary` specifies which Perl interpreter to use on the destination machine. (The Endeca software includes and requires version 5.8.3 of Perl.)
- `sslcertfile` specifies the path to the `eneCert.pem` certificate file on the destination machine.

The `source`, `dest`, `endeca_bin`, and `jcd_port` settings are required. The `sslcertfile` setting is required if the Endeca JCD is configured to use SSL. The `perl_binary` setting is optional but highly recommended.

Note: While it is not required, Endeca highly recommends that you specify `endeca_bin`, `jcd_port`, `perl_binary` and `sslcertfile` as global default settings.

The following is a UNIX example of a Fetch brick that uses the FTP protocol:

```
fetch_data : Fetch
 source = ftp://ftp.example.com/ourdata.zip
 username = endeca
 password = endeca
 dest = /raw_data/ourdata.zip
```

The following is a UNIX example of a Fetch brick that fetches data from a remote machine, using the Endeca JCD in a secure environment:

Note: This brick example assumes that `endeca_bin`, `jcd_port`, `perl_binary`, and `sslcertfile` have been set globally.

```
fetch_remote_data : Fetch
 source = \
   files://idx01/raw_data/ourdata.zip
 dest = /endeca/current/raw_data/ourdata.zip
```

Related Links

Global default settings reference on page 41
The tables below describe the most common control script settings that are set globally.

Shell brick

A Shell brick runs the operating system commands (DOS or shell) that you specify.

Shell bricks are frequently used to do pre- or post-processing, or for tasks for which no standard brick exists. Each line in a Shell brick is executed individually in sequence.

It is possible to write Shell bricks that run scripts that are external to the control script. If you write such a Shell brick, the external script it references must be stored locally on the machine on which it will be executed.

Shell brick settings

Setting	Description
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

The following is a UNIX example of a Shell brick called `arch01`. It removes the oldest archive file and rolls other versions back to make room for a new version.

```
arch01 : Shell
 rm /archive/4/*
 mv /archive/3/example1* /archive/4/
 mv /archive/2/example1* /archive/3/
 mv /archive/1/example1* /archive/2/
 ln /run/example1* /archive/1/
```

Forge brick

A Forge brick launches the Forge (Data Foundry) software, which transforms source data into tagged Endeca records.

Forge brick settings

Setting	Description
`pipeline`	Pathname of the `Pipeline.epx` file to pass to Forge, relative to the `working_dir`.
`dtd_dir`	Location where the DTD for the `Pipeline.epx` file resides. The default directory on Windows is `%ENDECA_ROOT%\conf\dtd`. The UNIX default is `/$ENDECA_ROOT/conf/dtd`.
`forge_binary`	Path to the Forge (Data Foundry) program.  **Note:** You can use this setting to override the `endeca_bin` default setting.

Setting	Description
`forge_options`	Command-line flags to pass to Forge.
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

The following is a Windows example of a Forge brick called `wine_forge`:

```
wine_forge : Forge
 forge_binary = $(endeca_root)\bin\forge.exe
 pipeline = Pipeline.epx
 forge_options = -n10000 -vw
```

Dgidx brick

A Dgidx brick sends the finished data prepared by Forge to the Dgidx program, which generates the proprietary indices for each MDEX Engine (Dgraph).

Dgidx brick settings

Setting	Description
`input`	The path to the Forge output.
`output`	The file prefix to use for generated files.
`dgidx_binary`	Path to the Dgidx indexing program.  **Note:** You can use this setting to override the `endeca_mdex_bin` default setting.
`dgidx_options`	Command-line flags to pass to Dgidx.
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

Setting	Description
run_aspell	Set to `true` (the default), runs the internal mechanism (called `aspell`). Set to `false`, `aspell` is not run.  **Note:** If you want to run the `dgwordlist` (UNIX) or `dgwordlist.exe` (Windows) script for spelling configuration (rather than `aspell`), you must do so separately in a Shell brick or elsewhere.

The following is a Windows example of a Dgidx brick called `dgidx01`:

```
dgidx01 : Dgidx
 dgidx_binary = C:\endeca\bin\dgidx.exe
 # path relative to working_dir
 input = .\forge_output\test.records.legacy
 output = .\indexed\test
```

AgraphIndex brick

The AgraphIndex brick runs Agidx on several Dgidx outputs, each of which can be stored on a different machine.

AgraphIndex brick settings

Setting	Description
`num_partitions`	Required. Specifies how many partitions will be combined. There must be one Dgidx output for each partition.
`output_path`	Required. Specifies where to put the Agidx output. This path must exist on each of the `partition`*N*`_machines`. The final output of the AgraphIndex brick will be placed on the last `partition`*N*`_machine`, in the location specified by this setting.
`partition0_machine`, `partition1_machine`, `...`	Required for all partitions from 0 to *num_partitions - 1*. Specifies the name of the Machine brick where the Dgidx output for a given partition is stored.
`partition0_path`, `partition1_path, ...`	Required for all partitions from 0 to *num_partitions - 1*. Specifies the path to the Dgidx output.
`endeca_mdex_bin`	Required. Set to the pathname of the `bin` directory in the Endeca MDEX Engine distribution you are using.

Setting	Description
`agidx_options`	Optional command-line flags to pass to Agidx.
`agidx_binary`	Optional path to the Agidx (aggregated indexing) program.  **Note:** You can use this setting to override the `endeca_mdex_bin` default setting.
`wget_binary`	Path to the wget file-retrieval program file (installed as part of the standard Endeca installation). This setting is optional; if it is not specified in the control script, the system looks for it in the `%ENDECA_ROOT%\utilities` directory on Windows and in `$ENDECA_ROOT/utilities` on UNIX.
`perl_binary`	Optional. Specifies which Perl interpreter to use on the destination machine. (The Endeca software includes and requires version 5.8.3 of Perl.)

In the following Windows example, AgraphIndex runs Agidx using the output of three Dgidx partitions:

```
agidx : AgraphIndex
 num_partitions = 3
 partition0_machine = idx00
 partition0_path = \
  C:\endeca\data\dgidx_output\partition0\dgidxout
 partition1_machine = idx01
  partition1_path = \
  C:\endeca\data\dgidx_output\partition1\dgidxout
 partition2_machine = idx02
 partition2_path = \
  C:\endeca\data\dgidx_output\partition2\dgidxout
 output_path = C:\endeca\data\agidx_output
```

Agidx brick

An Agidx brick runs Agidx on a machine, creating a set of Agidx indices that support the Agraph program in a distributed environment.

The Agidx brick is used only in distributed environments and is run sequentially on multiple machines. On the first machine, the Agidx brick takes the Dgidx output from that machine as its input. On the next machine, the output from the first Agidx run is copied over, using a Fetch brick. It, along with the Dgidx output from that machine, is used as Agidx brick input.

 **Note:** In many cases, a single AgraphIndex brick can take the place of several Agidx bricks, thus reducing overall script length.

Agidx brick settings

Setting	Description
`input`	The file prefix to the output of Dgidx on this machine.
`prev_output`	The file prefix of the Agidx data from the previous run, which has been copied to this machine by a Fetch brick. The `prev_output` setting is optional, and should not be used when running the Agidx brick on the first data subset.
`output`	The file prefix to the output of the Agidx run on this machine.
`agidx_binary`	Path to the Agidx (aggregated indexing) program.  **Note:** You can use this setting to override the `endeca_mdex_bin` default setting.
`agidx_options`	Command-line flags to pass to Agidx.
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

In the following Windows example, the results of an Agidx brick called `agidx01` are passed to `agidx02` and aggregated to that machine's Dgidx results:

```
agidx_all : Script
 agidx01
 copy_agidx01
 agidx02
 copy_agidx02

agidx01 : Agidx
 working_machine = idx01
 input = C:\endeca\data\idx\exmpl_dgidx01
 output = C:\endeca\data\idx\exmpl_agidx01

copy_agidx01 : Fetch
 working_machine = idx02
 source = \
  ftp://idx01//endeca/data/idx/exmpl_agidx01.*
 dest = C:\endeca\data\idx\

agidx02 : Agidx
 working_machine = idx02
 input = C:\endeca\data\idx\exmpl_dgidx02
 prev_output = C:\endeca\data\idx\exmpl_agidx01
 output = C:\endeca\data\idx\exmpl_agidx02

copy_agidx02 : Fetch
 working_machine = ag01
 source = \
  ftp://idx02//endeca/data/idx/exmpl_agidx02.*
 dest = C:\endeca\data\idx\
```

Note: In many cases, scenarios similar to the one shown in this example can be handled more concisely by using an AgraphIndex brick.

Related Links

AgraphIndex brick on page 74
The AgraphIndex brick runs Agidx on several Dgidx outputs, each of which can be stored on a different machine.

AgraphIndex brick on page 74
The AgraphIndex brick runs Agidx on several Dgidx outputs, each of which can be stored on a different machine.

Dgraph brick

A Dgraph brick runs the Dgraph (the MDEX Engine software).

Dgraph brick settings

Setting	Description
`input`	The file prefix for the output from Dgidx.
`port`	The port at which the Dgraph should listen. The default is 8000.
`startup_timeout`	Specifies the amount of time in seconds that the Control Interpreter will wait while starting the Dgraph. If it cannot determine that the Dgraph is running in this timeframe, it times out. The default is 60.
`dgraph_binary`	Path to the Dgraph (MDEX Engine) program. **Note:** You use this setting to override the `endeca_mdex_bin` default setting.
`dgraph_options`	Command-line flags to pass to Dgraph. **Note:** `--spellpath` is now automatically included unless you add it yourself and will be set to the directory containing the Dgidx output.
`log_dir`	Directory where the Dgraph writes its request log files.
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.

Setting	Description
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

Dgraph commands

The commands listed below can be used within Script bricks to control the Dgraph program:

Command	Description
`start`	Starts the Dgraph.  **Note:** This command succeeds even if the Dgraph is already started.
`running`	Succeeds if the Dgraph is running. This operation can be used to conditionalize a script (for example, "`if dgraph.running` …") or to check assumptions (like an assertion that the Dgraph is running at a certain time).
`stop`	Stops the Dgraph.  **Note:** This command succeeds even if the Dgraph is already stopped.
`update`	Checks for the presence of partial updates. If any partial update files are present, they are uploaded to the Dgraph.

This an example of a Dgraph brick called `dg01`:

```
dg01 : Dgraph
 working_machine = indexer
 port = 5555
 input = input.\indexed
```

The following excerpt from a Script brick illustrates how Dgraph commands work. This example shows how to stop and restart a running Dgraph:

```
restart_dg01 : Script
 dg01.stop
 dg01.start
```

Agraph brick

An Agraph brick runs the Agraph program, which defines and coordinates the activities of multiple, distributed Dgraphs.

Note: You can create child Dgraph bricks under an Agraph brick.

Agraph brick settings

Setting	Description
`input`	The Agidx outputs to aggregate, separated by spaces.
`port`	The port at which the Agraph should listen.
`children`	A list of the child Dgraphs for this Agraph, in the format *machine_name:port*.
`startup_timeout`	Specifies the amount of time in seconds that the Control Interpreter will wait while starting the Agraph. If it cannot determine that the Agraph is running in this timeframe, it times out. The default is 60.
`agraph_binary`	Path to the Agraph program. **Note:** You can use this setting to override the `endeca_mdex_bin` default setting.
`agraph_options`	Command-line flags to pass to Agraph.
`log_dir`	Directory where the Agraph writes its request log files.
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

Agraph commands

The commands listed below can be used within Script bricks to control the Agraph program:

Command	Description
`start`	Starts the Agraph but none of the Dgraphs.
`start_all`	Starts the Agraph and all its Dgraphs. This command only works if the Dgraphs are all child bricks (that is to say, indented) under the Agraph.

Command	Description
`running`	Succeeds if the Agraph is running. This command can be used to conditionalize a script (for example "`if agraph.running`") or to check assumptions, such as an assertion that the Agraph is running at a certain time.
`running_all`	Succeeds if the Agraph and all its Dgraphs are running. This command only works if the Dgraphs are all child bricks (that is, indented) under the Agraph.
`stop`	Stops the Agraph, but none of the Dgraphs.
`stop_all`	Stops the Agraph and all its Dgraphs. This command only works if the Dgraphs are all child bricks (that is, indented) under the Agraph.

The following is an example of an Agraph brick called `agraph01`:

```
agraph01 : Agraph
 working_machine = indexer
 port = 8888
 input = input.\indexed
 children = idx01:7777 idx02:7777 idx03:7777
```

The following excerpt from a Script brick stops `agraph01` and its three component Dgraphs.:

```
...
 agraph01.stop
 dgraph01.stop dgraph02.stop dgraph03.stop
```

Note: It is good practice (though not essential) to first stop an Agraph, and then stop its child Dgraphs. This is especially true in cases where the Agraph is actually serving requests from an end user.

Script brick

A Script brick tells the Control Interpreter how to execute the other bricks in the control script.

A Script brick consists of a list of other bricks to run. Script bricks may list other Script bricks, allowing you to build more complex Script bricks from simpler ones.

Most control scripts contain more than one Script brick. For example, a control script could have one Script brick that runs an entire data update sequence, and another that simply stops and restarts the Dgraph.

Script brick settings

Setting	Description
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

The following is an example of a Script brick called `run`:

```
run : Script
 archive01 archive02
 fetch01 fetch02
 dgidx01 dgidx02
 collect02 #copies dgidx output to a single machine
 agidx
 dgraph01.start dgraph02.start
 agraph.start
```

Implicit and explicit commands

Although most bricks within a Script brick have an implicit `run` command, some require explicit commands to use.

To execute bricks with the implicit command, the Script brick only has to list the brick in its definition. For example:

```
fetch_and_copy_data : Script
 fetch_data_1 fetch_data_2
 copy_data_1 copy_data_2
```

Three brick types, however, have explicit commands that you use to perform operations on them: Machine, Dgraph, and Agraph. Operations include things like starting, stopping, and testing the state of processes. In the sample Script brick below, a Dgraph brick called `dg01` is stopped and restarted.

```
restart_dg01 : Script
 dg01.stop
 dg01.start
```

Related Links

Machine brick on page 67
Machine bricks specify the name and connection details of each machine in a distributed environment.

Agraph brick on page 78
An Agraph brick runs the Agraph program, which defines and coordinates the activities of multiple, distributed Dgraphs.

Dgraph brick on page 77
A Dgraph brick runs the Dgraph (the MDEX Engine software).

Line execution

In a Script brick, by default, bricks named on different lines are run sequentially, while bricks named on the same line are run in parallel. You can also use explicit `parallel` and `sequential` lines to exercise additional control over line execution.

Lines that are indented beneath a `parallel` line run in parallel (note that this includes the first line of any nested `sequential` lines). Lines indented beneath a `sequential` line run sequentially. To illustrate these concepts, consider the Script brick below. Lines `first1` through `first5` will all start immediately, `second1` and `second2` will start as soon as `first4` is done, and `third` will start as soon as all the `firsts` and `seconds` are done. Note that `first2` and `first3`, and `second1` and `second2`, use the default syntax for parallel execution.

```
myscript : Script
 parallel
  first1
  first2 first3
  sequential
   first4
   second1 second2
  first5
 third
```

Note: If you have an `if` statement inside a parallel statement, the body of the `if` statement will execute sequentially, as if it were not inside the parallel statement.

Line-specific settings

Each line in a Script brick can have the following optional settings:

:

- `max_retry_count` — tells the Script brick how many times to retry a command if the first attempt fails. This setting does not include the Script brick's first attempt at running the command; in other words, a `max_retry_count` setting of "2" will result in a total of three attempts, the initial attempt plus two more.
- `retry_interval` — determines how long the Script brick waits, in seconds, before attempting to rerun a command that has failed. This setting is not useful unless the `max_retry_count` setting is also used.

If specified, the settings above should appear indented on the line following the command to which they apply. For example:

```
myscript : Script
 wait_for_something
  # Wait for one minute between attempts
  retry interval = 60
  # Try at most 3 times (including the first try)
  max_retry_count = 2
 do_something_else
```

if and else statements

Script bricks support if statements that allow you to conditionalize the execution of code.

An `if` statement runs a command. If that command succeeds, the code contained in the `if` statement is executed. If the command fails, the code inside the `if` statement is ignored:

```
myScript : Script
 if idx01.is_win32
  do_win32_version
```

`else` statements allow you to provide an alternative to the code contained in the `if` statement, in the event that the `if` command fails:

```
myScript : Script
 if idx01.is_win32
  do_win32_version
 else
  do_unix_version
```

try, onfail, and finally statements

Script bricks support three statements that allow you to exercise further control over the Control Interpreter in the event of an error:

- `try` — Defines a task that the Script brick attempts to accomplish.
- `onfail` — Defines what the Script brick should do if an error occurs during the `try` statement.
- `finally` — Defines what the Script brick should do after both the `try` and `onfail` statements have been executed.

Both `onfail` and `finally` statements are optional. However, you must include at least one of the two after a `try` statement. If an `onfail` statement exists, it is always executed before a `finally`.

If an error occurs during a `try` statement's execution, the Control Interpreter will proceed to an `onfail` statement, if one exists. The `onfail` allows you to do important tasks that should be finished despite the error. For example, if the `try` statement defines a data update sequence, you can use the `onfail` statement to restore the original data and restart your Dgraphs if the update fails.

Note: With `onfail`, the `try` statement's error is not hidden as it would be in an `if` statement.

The Control Interpreter always executes a `finally` statement, but not until after it executes the associated `try` and `onfail` (if there is one) statements.

It is important to understand how errors behave in a `try`/`onfail`/`finally` construct. If a `try` statement fails, the Control Interpreter considers it a fatal error, regardless of whether the `onfail` or `finally` statements succeed.

If a `try` statement succeeds, but a subsequent `finally` statement does not, the Control Interpreter reports the `finally` error as a non-fatal warning. `finally` errors do not cause the parent Script brick to fail.

Note: If a `try` statement succeeds, its associated `onfail` statement will not be executed.

If a `try` statement fails, and there are subsequent errors in the `onfail` or `finally` statements, the Control Interpreter reports the `try` statement's error as fatal, and the `onfail` and `finally` errors as non-fatal warnings. Again, `onfail` and `finally` errors do not cause the parent Script brick to fail.

Constants Brick

A Constants brick allows you to create named constants that are available for use by any of the other bricks in your control script.

The syntax for referencing constants looks like this:

`$(<constantsBrickName>.<constant>)`

The following is a Windows example of a Constants brick called `consts`:

```
consts : Constants
 data_dir = C:\Endeca\Data
```

To reference the `data_dir` constant defined above, you would use the following:

```
$(consts.data_dir)
```

For example:

```
dg01 : Dgraph
 input = $(consts.data_dir)\indexed
```

Archive brick

The Archive brick can create, archive, and roll back directories.

Archive brick settings

Setting	Description
`directory`	Required. Specifies the path to the directory that will be archived. For directories on different machines, specify a `working_machine`.
`max_archives`	Required. Specifies the maximum number of archives to store. This number does not include the original directory itself, so if `max_archives` was set to 3, you would have the original directory plus up to three archive directories, for a total of as many as four directories.
`archive_method`	Optional. Must be either move (the default) or copy.
`perl_binary`	Optional. Specifies the path to the Perl interpreter. Defaults to whatever Perl executable is in the path.

Archive brick commands

The commands listed below can be used within Script bricks to control the archiving process:

Command	Description
`create`	The create operation will first create the directory if it does not exist. In addition, it creates the `timestamp.txt` file if it doesn't exist. This file is used by the rollback operation to record when the directory was created. If the `timestamp.txt` file already exists, then it is left alone.
`archive`	Creates an archive directory from an existing directory. The archive directory will have the same name as the original directory, but with a timestamp appended to the end. The timestamp will reflect the time when the original directory was either created or last archived, not the time when the archive operation is performed. For example, if the original directory is called `logs` and was created on October 11, 2006 at 8:00 AM, the archive operation creates a directory called `logs.2006_10_11.08_00_00`.
`rollback`	Rolls back the directory to the most recent archived version. For example, say you have a directory called `logs`, one called `logs.2006_10_11.08_00_00`, and other, older versions. When you roll back, two things happen: • `logs` is renamed `logs.unwanted` • `logs.2006_10_11.08_00_00` is renamed `logs` The older versions are left alone. **Note:** There can only be a single `.unwanted` directory at a time. If you roll back twice, the `.unwanted` directory from the first rollback is deleted.

In the following UNIX example, an Archive brick is used to create a logs archive. It would be called from a Script brick or a RunCommand session using the command `logs.create`.

```
logs : Archive
 directory = /endeca/logs
 max_archives = 3
 archive_method = move
```

In this UNIX example, an Archive brick is used to make backup copies of state directories. It would be called using the `archive_state_MACHINE.archive` command:

```
archive_state_MACHINE : Archive
 working_machine = MACHINE
 directory = $(project_root)/state
 max_archives = 5
 archive_method = copy
```

Perl brick

The Perl brick allows you to incorporate custom Perl code into your control script.

Perl bricks have several settings, described below. The Control Interpreter considers any content other than these settings to be custom Perl code.

Note: Control Interpreter variables use a different syntax than Perl variables, so the two do not collide.

Perl brick settings

Setting	Description
`perl_binary`	The location of the Perl binary to be used.
`perl_options`	Any command line options you want to use when running the Perl binary.
`code`	A read-only setting that contains the body of the Perl brick. The Control Interpreter uses this setting to embed code from one Perl brick into another.
`stdout`	Where to redirect `stdout` for the brick. By default, `stdout` is sent to the screen. Specifying a value for `stdout` overrides the `stdout_base` setting.
`stderr`	Where to redirect `stderr` for the brick. By default, `stderr` is sent to the screen. Specifying a value for `stderr` overrides the `stderr_base` setting.

Pathnames in Perl bricks

Because of the way Perl treats escape characters, control script variables that contain Windows pathnames require special handling. To use a pathname in a Perl brick, put the path in single quotation marks (‘*<path>* ’) with a space at the end, then write an additional Perl statement that removes the space:

```
my $data_dir = ‘$(consts.data_dir) ‘;
$data_dir =~ s/ $//;
```

Reusing Perl bricks

You can embed the code from one Perl brick inside another Perl brick. To do this, you use the following statement:

```
perl_brick_2 : Perl
 ...other perl code...
 $(perl_brick_1.code)
 ...other perl code
```

where `perl_brick_1` is the name of the Perl brick that has the code to be embedded.

LogServer brick

The LogServer brick controls the use of the Endeca Log Server.

LogServer brick settings

Setting	Description
`port`	The port to listen on. The default value is 8002 (that is, the default Dgraph port plus two).
`log_file_prefix`	The file path prefix to use for log files.
`gzip`	If `true`, compresses the generated log files using gzip compression and appends a `.gz` suffix to the log file name.  **Important:** If you use this option, logs will not be written to disk as soon as they are received. That means that if the Log Server crashes unexpectedly, you may lose some log entries.

LogServer commands

The commands listed below can be used within Script bricks to control the Log Server:

Command	Description
`start`	Starts the Log Server.  **Note:** This command succeeds even if the Log Server is already started.
`running`	Succeeds if the Log Server is running. This operation can be used to conditionalize a script (for example, "`if logserver.running` …"), or to check assumptions, like an assertion that the Log Server is running at a certain time.
`stop`	Stops the Log Server.  **Note:** This command succeeds even if the Log Server is already stopped.
`roll`	Causes the Log Server to close its log and open a new one with a new timestamp.

In the following Windows example, the Log Server is launched:

```
log_server : LogServer
  port = 8002
  log_file_prefix = C:\endeca\logs\log
```

ReportGenerator brick

The ReportGenerator brick runs the Report Generator, which processes Log Server files into HTML-based reports that you can view in your Web browser and XML reports that you can view in Endeca Workbench.

ReportGenerator brick settings

Setting	Description
`logs`	Required. Path to the file or directory containing the logs to report on. If it is a directory, then all log files in that directory are read. If it is a file, then just that file is read.
`output`	Required. What to name the generated report file. For example: output = `C:\Endeca\reports\myreport.html` on Windows output = `/endeca/reports/myreport.html` on UNIX
`settings`	Path to the `report_settings.xml` file. For example: `$(sample_wine_data_dir)\etc\report_settings.xml`
`stylesheet`	Required. Path to the `report_stylesheet.xsl` file. For example: `$(sample_wine_data_dir)\etc\report_stylesheet.xsl`
`timerange <`*`keyword`*`>`	Optional. Set the time span of interest (or report window). Allowed keywords: • `yesterday` • `last-week` • `last-month` • `day-so-far` • `week-so-far` • `month-so-far` These keywords assume that days end at midnight, and weeks end on the midnight between Saturday and Sunday.
`start_date <`*`date`*`>` `stop_date <`*`date`*`>`	Optional. These set the report window to the given date and time. The date format should be either `yyyy_mm_dd` or `yyyy_mm_dd.hh_mm_ss`. For example, `2010_03_25.19_30_57` expresses March 25, 2010 at 7:30:57 in the evening.
`time-series <`*`frequency`*`>`	Generates time series data at a specified frequency. The value can be either `hourly` or `daily`.

Setting	Description
`charts <status>`	Generates charts in reports. The value is either `enable` or `disable`. If unspecified, the default is set to `disable`.
`java_binary`	Optional. Should indicate a JDK 1.5.x or later. Defaults to the JDK that Endeca installs.
`java_options`	Optional. Command-line options for the `java_binary` setting. This command is primarily used to adjust the report generator memory, which defaults to 1GB. To set the memory, use the following (ignore the linebreak): `java_options = -Xmx[MemoryInMb]m -Xms[MemoryInMb]m`

This Windows example of a ReportGenerator brick generates a report for all logs in a directory:

```
    wine_genreport : ReportGenerator
       logs   = $(sample_wine_data_dir)\logs\logserver_output\
       settings   = $(sample_wine_data_dir)\etc\report_settings.xml
       stylesheet  = $(sample_wine_data_dir)\etc\report_stylesheet.xsl
       output   = $(sample_wine_data_dir)\reports\sample_report.html
```

Example control script

This simple but complete control script demonstrates how all of the Control Interpreter elements work together.

The control script is based on the reference implementation that is part of the Endeca installation, and is designed to run on a single machine. This example is for a Windows environment.

```
############################## Global Variables ################################
#### Global variables can be reused anywhere in the control script, and
#### certain global variables (such as jcd_port and working_dir) are used
#### as defaults for the control interpreter.

# JCD connection
jcd_port = 8088
jcd_use_ssl  = false

# Dgraph and LogServer ports
#
# IMPORTANT NOTE: Although not required, Reference UI expects logserver to
 be
# running on dgraph port +2
#
dgraph_port  = 8000
logserver_port  = 8002

# Reusable path variables
sample_wine_data_dir = $(endeca_root)..\reference\sample_wine_data
```

```
# Common operational variables for all bricks
working_machine = wine_indexer
working_dir     = $(sample_wine_data_dir)\data
stdout_base     = ..\logs\out.
stderr_base     = ..\logs\err.

# Location of Perl 5.8.3 binary (required for Fetch brick)
perl_binary = $(endeca_root)\perl\5.8.3\bin\perl.exe

############################## Bricks ##########################################
#### Bricks define interfaces to various programs.
####
#### Endeca components such as forge, dgidx and dgraph have special bricks
#### that know about the process they are running.
####
#### For other user-defined actions, a Shell brick can be used to run any
#### system command.

# This brick defines the machine to be used for the data update process.
wine_indexer : Machine
    name = localhost

# This brick runs forge to process the raw data.
wine_forge : Forge
    pipeline = ..\data\forge_input\pipeline.epx
    forge_options = -vw

# This brick runs dgidx to index the processed data.
wine_dgidx : Dgidx
    input  = ..\data\partition0\forge_output\wine
    output = ..\data\partition0\dgidx_output\wine

# This brick moves index files from dgidx_output to dgraph_input.
wine_fetch : Fetch
   source = file:///$(sample_wine_data_dir)\data\partition0\dgidx_output\*

    dest = .\partition0\dgraph_input
    remove_source = true

# This brick runs the dgraph, using the indices created by the
# wine_dgidx brick. Note that the global setting for working_dir
# has been overridden in this brick.
wine_dgraph : Dgraph
    working_dir = $(sample_wine_data_dir)\logs
    input = ..\data\partition0\dgraph_input\wine
    port = $(dgraph_port)

# This brick runs logserver to handle application logging requests.
wine_logserver : LogServer
    port            = $(logserver_port)
    log_file_prefix = $(sample_wine_data_dir)\logs\logserver_output\wine




# This brick generates an html report for all logs in a directory.
wine_genreport : ReportGenerator
    logs = $(sample_wine_data_dir)\logs\logserver_output\
    settings = $(sample_wine_data_dir)\etc\report_settings.xml
    stylesheet = $(sample_wine_data_dir)\etc\report_stylesheet.xsl
```

```
    output = $(sample_wine_data_dir)\reports\sample_report.html

# This brick generates an xml report for use by the business studio
wine_toolsreport : ReportGenerator
   logs = $(sample_wine_data_dir)\logs\logserver_output\
   settings = $(sample_wine_data_dir)\etc\report_settings.xml
   stylesheet = $(sample_wine_data_dir)\etc\tools_report_stylesheet.xsl
   output = $(sample_wine_data_dir)\reports\tools_report.xml
   timerange = "day-so-far"


############################## Scripts ########################################
#### Scripts are called to run each brick in the correct order to accomplish
#### tasks.

# This Script brick runs the entire data update sequence: process
# the data using forge, index the data using dgidx, stop the
# currently running dgraph and restart it using the new indices.
runme : Script
    wine_forge
    wine_dgidx
    if wine_dgraph.running
      wine_dgraph.stop
    wine_fetch
    wine_dgraph.start
    if wine_logserver.running
      wine_logserver.stop
    wine_logserver.start

# This Script brick stops and restarts the dgraph. This brick
# is useful in cases where the administrator wants to restart
# the dgraph but does not need to rerun forge or dgidx.
dgraph_start : Script
    if wine_dgraph.running
      wine_dgraph.stop
    wine_dgraph.start

# This Script brick stops the dgraph if it is running.  This
# is useful when re-defining bricks on a running JCD
dgraph_stop : Script
    if wine_dgraph.running
      wine_dgraph.stop
# This Script brick stops and restarts the logserver. This brick
# is useful in cases where the administrator wants to restart
# the logserver but does not need to rerun forge or dgidx or restart
# the dgraph.
logserver_start : Script
    if wine_logserver.running
      wine_logserver.stop
    wine_logserver.start

# This Script brick rolls the log file generated by the LogServer.
logserver_roll : Script
    wine_logserver.roll
```

Appendix B

Control System-based Examples

This appendix contains examples of control scripts that are based on using the Control System.

Control scripts and term extraction pipelines

You can run the pipeline for term discovery with either the Endeca Application Controller (EAC) or control scripts.

For more information on the term discovery pipeline, see the *Endeca Relationship Discovery Guide*.

The only recommended practice when running the pipeline with control scripts is to use the Forge `--stateDir` flag to specify the location of the state directory. For example, you can set a global variable to the location of the state directory:

```
state_dir = $(sample_te_data_dir)\data\partition0\state
```

You then use that variable as the argument to the `--stateDir` flag in the Forge brick:

```
baseline_forge : Forge
  pipeline = ..\data\forge_input\pipeline.epx
  forge_options = -vw --stateDir $(state_dir)
```

Otherwise, there is nothing different about a control script that is used to support term discovery.

Control scripts in differential crawling

Differential crawling cannot be run from Developer Studio nor from Endeca Workbench. It can be run only via a control script.

A sample control script is listed in this chapter. For information on the pipeline used for this script, see the *Endeca Forge Guide*.

The control script uses two high-level Script bricks (`full_crawl` and `diff_crawl`) to implement the two crawling processes.

Note: In order to illustrate the basics of a differential crawling control script, the sample control script does not have a full range of Endeca features, such as a brick for the logging server.

Related Links

Sample control script for differential crawling on page 95

A sample differential crawling control script is presented below:

About the differential crawling script

You can run a differential crawl using a high-level Script brick.

In the control script, the Script brick, `diff_crawl`, implements the differential crawling procedure as follows:

```
diff_crawl : Script
 differential_forge
 differential_dgidx
 differential_fetch
 if differential_dgraph.running
  differential_dgraph.stop
 differential_dgraph.start
```

You run a differential crawl with a command line similar to this Windows example (assuming you are in the `etc` directory):

```
runcommand control.script diff_crawl
```

The four called bricks are very straightforward. For example, the `differential_forge` brick runs Forge on the source data that is incoming from the spider component, using these settings:

```
pipeline = ..\data\forge_input\pipeline.epx
forge_options = -vw
```

You may want to modify the `forge_options` setting so that it is better suited for your application. Note, however, that no special Forge flags are needed to process the Endeca Crawler's output.

About the full crawling script

The `full_crawl` Script brick removes previous crawler output, then runs a full crawl.

In the control script, the high-level Script brick, `full_crawl`, implements the full crawling procedure as follows:

```
full_crawl : Script
 differential_clearcrawlstate
 diff_crawl
```

You run this script with a command line similar to this Windows example (assuming you are in the `etc` directory):

```
runcommand control.script full_crawl
```

The two steps of the `full_crawl` script are as follows:

1. Delete the previous crawler output

Before a full crawling procedure is run, the `differential_clearcrawlstate` Shell brick first removes the previous crawler output by deleting two files in the state directory:

- `previouscrawl.records.binary` — contains the records of the previous crawl.
- `differential_state.gz` — contains the record metadata of the previous pipeline run.

2. Run the diff_crawl script

The `full_crawl` brick then calls the `diff_crawl` brick to run a full crawl, run Forge on the resulting data, run Dgidx, and start the MDEX Engine.

Sample control script for differential crawling

A sample differential crawling control script is presented below:

```
# Reference control script to demonstrate differential crawling using
# the Endeca Crawler.
# Copyright (c) 2006, Endeca Technologies, Inc.

########################## Global Variables ##############################
# JCD connection
jcd_port = 8088
jcd_use_ssl = true
sslcertfile = $(endeca_root)\..\workspace\etc\eneCert.pem

# Reusable path variables
sample_differential_data_dir = C:\Projects\sample_differential_data

# Common operational variables for all bricks
working_machine = differential_indexer
working_dir     = $(sample_differential_data_dir)\data
stdout_base     = ..\logs\out.
stderr_base     = ..\logs\err.

perl_binary = $(endeca_root)\perl\5.8.3\bin\perl

########################## Bricks ########################################
# Defines the machine used for the data update process.
differential_indexer : Machine
 name = localhost

# Runs Forge to process the raw data, including crawling.
differential_forge : Forge
 pipeline         = ..\data\forge_input\Pipeline.epx
 forge_options    = -vv

# Runs Dgidx to index the processed data.
differential_dgidx : Dgidx
 input         = ..\data\partition0\forge_output\differential
 output        = ..\data\partition0\dgidx_output\differential
 dgidx_options = -v

# Moves index files from dgidx_output to dgraph_input
differential_fetch : Fetch
 source = file:///$(sample_differential_data_dir)\data\partition0\dgidx_out¬
put\*
 dest = \partition0\dgraph_input
 remove_source = true

# Runs the Dgraph, using the indices created by the
# differential_dgidx brick. Note that the global setting for working_dir
# has been overridden in this brick.
differential_dgraph : Dgraph
 working_dir = $(sample_differential_data_dir)\logs
 input       = ..\data\partition0\dgraph_input\differential
```

```
 port        = 8000

# Removes previous crawler output in order to run a new full crawl
differential_clearcrawlstate : Shell
 $(perl_binary) $(endeca_root)/bin/utility_cmds.pl rmFile $(sample_differ¬
ential_data_dir)/data/partition0/state/previouscrawl.records.binary
 $(perl_binary) $(endeca_root)/bin/utility_cmds.pl rmFile $(sample_differ¬
ential_data_dir)/data/partition0/state/differential_state.gz

########################## Scripts ###################################

# This Script brick runs the entire data update sequence: process
# the data using forge, index the data using dgidx, stop the
# currently running dgraph and restart it using the new indices.
diff_crawl : Script
 differential_forge
 differential_dgidx
 differential_fetch
 if differential_dgraph.running
  differential_dgraph.stop
 differential_dgraph.start

# This Script performs a full crawl by deleting previous data
# and running the diff_crawl script.
full_crawl : Script
 differential_clearcrawlstate
 diff_crawl
```

About using control scripts for baseline and partial updates

This section describes control script development and execution for baseline updates and partial updates.

For more information on baseline updates, see the *Endeca Forge Guide*. For information on partial updates, see the *Endeca Partial Updates Guide*.

Sample control script for partial updates

A sample partial update control script is presented below:

```
# Reference control script to run baseline updates and partial updates.
# Copyright (c) 2007, Endeca Technologies, Inc.
#
########################## Global Variables ##################################

jcd_port = 8088
jcd_use_ssl = true
sslcertfile = $(endeca_root)\..\workspace\etc\eneCert.pem
dgraph_port = 8000

sample_updates_data_dir =
$(endeca_root)\..\reference\sample_updates_data
update_dir = ..\data\partition0\dgraph_input\updates\
working_machine = indexer
working_dir = $(sample_updates_data_dir)\data
stdout_base = ..\logs\out.
stderr_base = ..\logs\err.
```

```
perl_binary = $(endeca_root)\perl\5.8.3\bin\perl.exe
wget_binary = $(endeca_root)\utilities\wget

########################## Bricks ########################################

indexer : Machine
 name = localhost

dgraph : Dgraph
 working_dir = $(sample_updates_data_dir)\logs
 input       = ..\data\partition0\dgraph_input\wine
 port        = $(dgraph_port)
 dgraph_options =  --updatedir $(update_dir) --updateverbose

clear_updates: Shell
 del /f /q $(sample_updates_data_dir)\data\partition0\dgraph_input\updates\*

########################## Baseline Update Bricks ########################

baseline_forge : Forge
 pipeline         = ..\data\forge_input\pipeline.epx
 forge_options    = -vw

baseline_dgidx : Dgidx
 input  = ..\data\partition0\forge_output\wine
 output = ..\data\partition0\dgidx_output\wine
baseline_fetch : Fetch
 source = file:///$(sample_updates_data_dir)\data\partition0\dgidx_output\*

 dest = .\partition0\dgraph_input
 remove_source = true

########################## Partial Update Bricks #########################

update_forge : Forge
 pipeline         = ..\data\forge_input\partial_pipeline.epx
 forge_options    = -vw

apply_timestamp : Shell
 $(perl_binary) ..\etc\applytimestamp.pl ..\data\partition0\dgraph_input\up¬
dates\wine-sgmt0.records.xml

########################## Scripts ########################################

# This script runs a baseline update, runs Dgidx, and starts the Dgraph.
baseline_update : Script
 clear_updates
 baseline_forge
 baseline_dgidx
 if dgraph.running
  dgraph.stop
 baseline_fetch
 dgraph.start

# This script runs a partial update and restarts the Dgraph with the
# modified data.
partial_update : Script
 update_forge
 apply_timestamp
 if dgraph.running
```

```
  dgraph.update

# This script restarts the Dgraph.
restart : Script
 if dgraph.running
  dgraph.stop
 dgraph.start
```

Directory structure for updates

The directory structure for updates is outlined below, with an overview of each directory's purpose.

The control script uses the following directory structure for handling data flow through the system:

```
data
 forge_input
 incoming
   updates
 partition0
   dgidx_output
   dgraph_input
     updates
   forge_output
   state
```

Directory	Purpose
`data`	Base directory for all other subdirectories. All files and processes related to the data exist and work in or under this directory.
`data\forge_input`	Contains the Developer Studio project file (`sample_updates.esp`), the baseline update pipeline file (`pipeline.epx`), the partial update pipeline file (`partial_pipeline.epx`), and the index configuration files (`*.xml`).
`data\incoming`	Contains source data (in the `wine_data.txt.gz` file) for a baseline update. On a production site, the files in this directory may have been created by a data extraction process on the customer's database or may be picked up from another FTP server.
`data\incoming\updates`	Contains source data for a partial update. The control script assumes that the directory has three gzipped files: `adds.txt.gz` (records to be added), `deletes.txt.gz` (records to be deleted), and `up¬dates.txt.gz` (records to be updated).
`data\partition0`	Contains files generated by the Forge, Dgidx, and Dgraph programs.
`data\partition0\dgidx_output`	Contains indices that have been processed by Dgidx and output in a format that can be read by the MDEX Engine.

Directory	Purpose
data\partition0\dgraph_input	Contains data that is read by the MDEX Engine on startup. The data includes the Dgidx output indices, spelling correction dictionaries, thesaurus files, and language-encoding files.
data\partition0\dgraph_input\updates	Contains partial updates that have been processed by Forge. The MDEX Engine reads these updates when it is restarted with the Dgraph `--updatedir` flag pointing to this directory.
data\partition0\forge_output	Contains data that has been processed by Forge and is ready for indexing.
data\partition0\state	Contains any state information (such as auto-generated dimension IDs) that must be saved between Forge runs.

Note: All references to directory names in the following text are relative to the data directory. All references to directory names in example or default brick definitions are relative to the parent of the data directory.

About the baseline updates script

The `baseline_update` Script brick calls other Script bricks to delete old updates, run Forge and Dgidx, and restart the MDEX Engine with updated index files.

In the control script, the high-level Script brick, `baseline_update`, implements the baseline update procedure by making calls to other Script bricks:

```
baseline_update : Script
 clear_updates
 baseline_forge
 baseline_dgidx
 if dgraph.running
  dgraph.stop
 baseline_fetch
 dgraph.start
```

You run a baseline update with a command line similar to this Windows example (assuming you are in the control script's directory):

```
runcommand update_index.script baseline_update
```

The baseline update process is as follows:

1. Delete old updates

All files in the `data\partition0\draph_input\updates` directory are deleted by the `clear_updates` brick.

2. Run Forge

The `baseline_forge` brick runs Forge on the source data, using these default settings:

```
The baseline_forge brick runs Forge on the source data, using these default
 settings:
pipeline = ..\data\forge_input\pipeline.epx
forge_options  = -vw
You will want to modify the forge_options setting so that it is better
suited for your application.
```

You will want to modify the `forge_options` setting so that it is better suited for your application.

3. Run Dgidx

The `baseline_dgidx` brick runs Dgidx with these settings:

```
input = ..\data\partition0\forge_output\wine
output = ..\data\partition0\dgidx_output\wine
```

You will probably want to modify the options passed to Dgidx. You will also want to change:

- The `input` setting so that it points to the location where your pipeline writes out the Forge output data.
- The `output` setting so that it points to the location where Dgidx should write out data for the MDEX Engine (make sure that the location ends with the prefix that you want to use for the Dgidx output).

4. Stop the MDEX Engine

The `dgraph.stop` command stops the MDEX Engine.

5. Move the index files to the Dgraph directory

The `baseline_fetch` brick moves index files from the `data\partition0\dgidx_output` directory to the `data\partition0\dgraph_input` directory, where they are used by the MDEX Engine on startup.

Be sure to change the paths in the `source` and `dest` settings for your implementation.

6. Start the MDEX Engine

The MDEX Engine is started with the `dgraph` brick, using these settings:

```
working_dir = $(sample_updates_data_dir)\logs
input       = ..\data\partition0\dgraph_input\wine
port        = $(dgraph_port)
dgraph_options = --updatedir
       ..\data\partition0\dgraph_input\updates
```

You may want to use the `--updateverbose` flag during development, but make sure you remove it for production. You may want to add other options relevant for your application. See the *Endeca IAP Administrator's Guide* for information about the available Dgraph options.

At this point, the MDEX Engine should be running correctly with the latest baseline and partial update data.

About the partial updates script

The `partial_update` Script brick processes records with Forge, then applies a timestamp and restarts the MDEX Engine with the updated indexes.

In the control script, the high-level Script brick, `partial_update`, implements the partial update procedure as follows:

```
partial_update : Script
 update_forge
 apply_timestamp
 if dgraph.running
   dgraph.update
```

You run a partial update with a command line similar to this Windows example (assuming you are in directory where the control script resides):

```
runcommand update_index.script partial_update
```

The three major steps of the `partial_update` Script brick are described below:

1. Run Forge on the new source data

The `update_forge` brick runs Forge with the partial update pipeline and new source data, using these default settings:

```
pipeline = ..\data\forge_input\partial_pipeline.epx
forge_options = -vw
```

Because the record adapter uses the Multi Files setting, Forge can read data from multiple input files. (This implementation uses three input files.)

You will want to modify the `forge_options` setting so that it is better suited for your application. Modify the relative paths above as appropriate for your implementation.

When Forge finishes, it produces one or more update record files and stores them in the location specified by the pipeline's update adapter. This file contains XML definitions of how the updated records should be treated by the MDEX Engine (for example, which records to delete or add).

The record files use this naming format:

db_prefix-sgmtn.records.xml

For example, the `update_forge` brick outputs the `wine-sgmt0.records.xml` file in the `data\partition0\dgraph_input\updates` directory.

The `-sgmt0` portion of the filename is generated when you roll over by size (i.e., the update indexer contains the `ROLLOVER` element, as in the partial updates pipeline). Forge splits the output into segment files, each of which is no larger than 2GB.

Note: It is important that you know the names of the record files, because they will have to be timestamped, as described in the next section.

2. Apply a timestamp to the record file

It is possible to generate multiple partial updates before the next baseline update, at which time all the partial update files are deleted. Therefore, each record file must be timestamped to ensure that the MDEX Engine does not upload a partial update more than once.

The `apply_timestamp` brick renames the `db_prefix-sgmtn.records.xml` files by appending a timestamp string to the filename. The resulting filename will use this format:

originalfilename_YYYY.MM.DD.HH.NN.SS

where *YYYY* is the four-digit year, *MM* is the two-digit month, *DD* is the two-digit day, *HH* is the two-digit hour, *NN* is the two-digit minute, and *SS* is the two-digit second. For example:

```
wine-sgmt0.records.xml_2010.03.07.16.14.08
```

A running MDEX Engine keeps track of the last timestamped file it uploaded. When it next checks the updates directory, it will only upload partial update files that carry a timestamp later than the last uploaded file.

Note: The `apply_timestamp` brick in the control script assumes that only one record file will be renamed. If your implementation generates multiple record files, you will need to change this brick for the additional renaming statements.

3. Update the MDEX Engine

The `dgraph.update` command causes the running MDEX Engine to perform the following actions:

1. Go offline while it processes the updates (that is, it stops accepting user queries and temporarily closes its listening port).
2. Check the updates directory (whose path is specified with the `--updatedir` flag).
3. Upload any partial update with a timestamp later than the last currently-loaded partial update.
4. Go back online after it has processed all updates.

At this point, the MDEX Engine should be running correctly with the latest baseline and partial update data.

About adding other bricks

You can modify the `update_index.script` and add other bricks that are necessary for your implementation.

For example, you can add a brick that fetches partial updates from an FTP server. In other installations, the partial updates may be dropped onto the indexing server, directly into the `incoming\updates` directory.

The following is an example of a Fetch brick:

```
fetch_updates : Shell
 perl bin/fetch.pl \
  --ftp_ip ftp.somecompany.com \
  --ftp_user anonymous \
  --ftp_pass somecompany.com \
  --fetch_dir incoming/ \
  --fetch_file_regexp
    "endeca_update_200407(\d+)\.txt" \
  --exclude_file etc/exclude_files \
  --dest_dir data/incoming/updates
```

The flags in the example are:

- `--ftp_ip` — the IP address of the FTP server.
- `--ftp_user` — the username for logging into the FTP server.
- `--ftp_pass` — the password for the username.
- `--fetch_dir` — the directory on the FTP server that contains the update files to retrieve.
- `--fetch_file_regexp` — the regular expression that should be matched for a file to be considered a partial update file.
- `--exclude_file` — points to a file that will be maintained automatically by `fetch.pl`. It is a list of all the files that have already been retrieved from the FTP server and should not be retrieved again.

- `--dest_dir` — the directory into which the fetched files will be dropped.

Related Links

Control Script Brick Reference on page 67

This appendix provides details about specific brick types. Some long brick settings break onto the following line in these examples; however, you should type each brick setting on a single line. If you need to wrap a line, put a space and a backslash (" \") at the break; this tells the Control Interpreter to ignore the line break.

The Dgraph update command

An update command can be issued to the MDEX Engine in one of two ways:

- With a URL command in your browser. This method allows you to specify options to the command. For information, see the *Endeca Forge Guide*.
- From a control script. With this method, you cannot specify options.

Related Links

The Dgraph update command in control scripts on page 103

The `dgraph.update` command cannot specify options when used from a control script.

The Dgraph update command in control scripts

The `dgraph.update` command cannot specify options when used from a control script.

The `dgraph.update` command can be issued from a control script, as in this example:

```
run_partial_update : Script
 update_forge
 apply_timestamp
 if dgraph.running
  dgraph.update
```

When the `run_partial_update` script is executed, the `dgraph.update` command is issued to the MDEX Engine to begin processing the partial update files.

The behavior of this version of the command is identical to the default behavior of the URL version. The only difference between the two versions is that the `dgraph.update` control script version cannot take the `offline` and `warmupseconds` options.

About using a control script for Agraph updates

This section contains information about using control scripts for running partial updates for configurations that contain an Agraph.

For information on building an Agraph pipeline to run the partial update, see the *Endeca Partial Updates Guide*.

The sample control script implements partial updates for a single-machine, single-Dgraph deployment only. For an Agraph deployment, you can modify the control script to run Forge on a single machine and distribute the Forge output to all the other machines. Then, you notify each Dgraph in your deployment to check for new updates.

Forge partial updates brick

The partial updates Forge brick is similar to the `update_forge` brick, but with a recommended additional flag.

For a Forge brick that processes partial update source data, Endeca recommends the use of the Forge `--numPartitions` flag to specify the number of Agraph partitions:

```
# Runs Forge on the update source data.
update_forge : Forge
  working_machine = indexer
  pipeline = ..\data\forge_input\partial_pipeline.epx
  forge_options = -vw --numPartitions $(numPartitions)
```

Using the `--numPartitions` flag (which overrides the `NUM_IDX` setting in the update adapter) lets you easily add or subtract Agraph partitions from within the control script. You will have to set up a global variable (named *numPartitions* in the example above) that stores the number of partitions.

Related Links

About the partial updates script on page 100
The `partial_update` Script brick processes records with Forge, then applies a timestamp and restarts the MDEX Engine with the updated indexes.

About distributing the Forge output to Dgraphs

For a deterministic distribution strategy, the distribution of the record files depends on the use of auto-generated dimensions.

For a random distribution strategy, partial updates in Agraph implementations do not require any special update distribution requirements. Both dimension modifications (i.e., dimension value additions) and record modifications (updates, deletes, replaces, and adds) should be sent to all Dgraphs in the deployment. Each Dgraph should then be notified to check for new updates. If a Dgraph cannot handle data that is associated with another Dgraph, it will simply log a warning but will otherwise continue working. Note that the Agraph process itself does not process updates.

A deterministic distribution needs to be configured differently:

- If you are using auto-generated dimensions, distribute all the record files to all the Dgraphs.
- If you are not using auto-generated dimensions, you can distribute each record file to its specific Dgraph.

To make sure that there is no interruption in servicing navigation requests, you may configure your Dgraphs to check for new updates at different times. Or you can also have smaller subgroups read in updates simultaneously (for example, three machines at a time in a six-machine implementation).

Using control scripts with the Agraph

The following planning considerations apply if you are using control scripts with the Agraph.

For information about Agraph, see the *Endeca Advanced Developer's Guide*.

Note: This section does not apply to Analytics implementations.

Take the following actions when using control scripts with the Agraph:

1. Arrange your partitions and data files so that they are available to the various Dgidx processes.

When running your implementation with a control script, you have to arrange data files so that they are available to the various Dgidx processes. In particular, each Dgidx process needs to access its corresponding partition of the records, as well as the configuration files that are common to all of the processes. If the Dgidx processes are to be executed on the different machines, then the control script must distribute files across machines.

2. Shut down the Agraph during dynamic business rules updates.

   In a control script environment, Endeca recommends that you shut down the Agraph during dynamic business rule updates. (In an environment that uses EAC and tools, the Agraph automatically shuts down during any type of update process and then restarts after the update completes).

   If you do not shut down the Agraph during the update, end-users will not receive a response to requests made during this short update time and the Agraph issues a fatal error similar to the following:

   ```
   [Thu Mar 23 16:26:29 2006] [Fatal] (merchbinsorter.cpp::276) -
   Dgraph 1 has fewer rules fired.
   ```

Appendix C

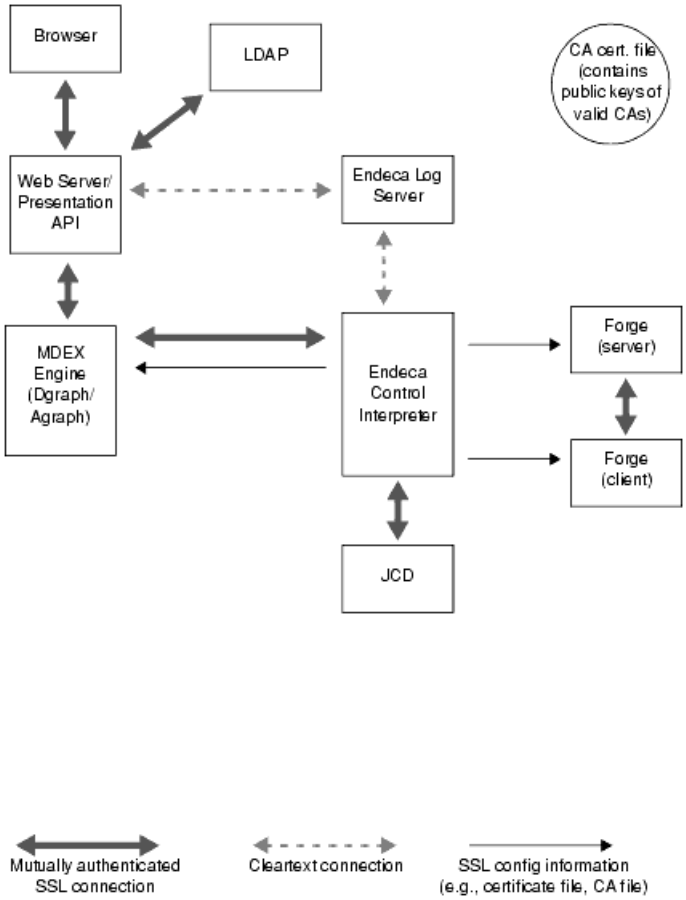
SSL Configuration for the Control Interpreter

This appendix describes how to use SSL with the Control Interpreter.

Control Interpreter system communications

Endeca components use different methods of communication, and not all of them can use SSL.

The illustration below shows communication methods between components of an Endeca secure implementation when using the Control Interpreter:



As the illustration shows, not all components can use SSL. For example, the Endeca Log Server currently cannot be configured to use SSL.

Note: For general information about configuring your Endeca implementation to use SSL, see the *Endeca Security Guide*.

Enabling SSL for the MDEX Engine and Forge

This section describes configuring the MDEX Engine to use SSL and, optionally, mutual authentication when communicating with the Presentation API and other Endeca system components.

Note: This section assumes that you configured the Endeca JCD service to use SSL when communicating with the Control Interpreter.

You can configure Forge to use SSL between the Forge server and Forge client in a parallel Forge implementation. The configuration process depends on whether you are using a Control Interpreter script or the Endeca Application Controller. Both processes are described in the following sections.

Keep in mind that you will be using the following two certificate files to configure SSL:

- `eneCert.pem` — The certificate used by all clients and servers to specify their identity when using SSL. This certificate file should be thought of as the identity of the Endeca system, or as the identity of all components of the Endeca system.
- `eneCA.pem` — The Certificate Authority (CA) file used by all clients and servers that wish to authenticate the other endpoint of a communication channel.

These certificate files are not shipped with the product. Therefore, you must use the enecerts utility to generate them, as described in the *Endeca Security Guide*.

Related Links

Enabling authentication and security on page 19
If your implementation requires it, the Endeca JCD can authenticate the identity of all client requests through the use of SSL certificates. You need to generate a set of certificate files to enable SSL.

Control Interpreter script configuration

If you are using a Control Interpreter script to run the Forge and Dgraph/Agraph programs, you can specify their SSL settings either as global default settings or in their bricks.

Endeca recommends specifying SSL settings as global defaults, and only specifying SSL settings within individual bricks when you want to override a global default for a particular brick.

In the following sections, it is assumed that the JCD has been configured to use SSL by using the `jcd_use_ssl` setting.

Related Links

Global default settings reference on page 41
The tables below describe the most common control script settings that are set globally.

Dgraph/Agraph SSL settings

The following table lists the SSL-specific settings for the Dgraph and Agraph bricks:

Dgraph/Agraph brick setting	Description
`ene_use_ssl`	If set to `true`, specifies that all MDEX Engines (Dgraphs or Agraphs) will use SSL when communicating with other Endeca system components. If set to `true`, you must also use the `sslcertfile` setting.
`sslcertfile`	Specifies the path of the `eneCert.pem` certificate file that will be used by the MDEX Engine (Dgraph/Agraph) processes to present to any client. This is also the certificate that the Control Interpreter should present to the MDEX Engine when trying to talk to the MDEX Engine as a client.
`sslcafile`	Specifies the path of the `eneCA.pem` Certificate Authority file that the MDEX Engine (Dgraph/Agraph) processes will use to authenticate communications with other Endeca components. If `ene_use_ssl` is set to `true`, using `sslcafile` will turn on authentication.

Dgraph/Agraph brick setting	Description
`sslcipher`	A cipher string (such as `RC4-SHA`) that specifies the minimum cryptographic algorithm that the Dgraph/Agraph processes will use during the SSL negotiation. If you omit this setting, the SSL software will try an internal list of ciphers, beginning with `AES256-SHA`. See the *Endeca Security Guide* for a list of cipher strings.

Forge SSL settings

The following table lists the SSL-specific settings for the Forge brick:

Forge brick setting	Description
`forge_use_ssl`	If set to `true`, specifies that Forge clients and servers will use SSL to communicate with each other when running in parallel Forge mode. If set to `true`, you must also use the `sslcertfile` setting.
`sslcertfile`	Specifies the path of the `eneCert.pem` certificate file that will be used by the Forge server and Forge clients.
`sslcafile`	Specifies the path of the `eneCA.pem` CA file that the Forge server and Forge clients will use to authenticate each other. If `forge_use_ssl` is set to `true`, using `sslcafile` will turn on authentication.
`sslcipher`	A cipher string (such as `RC4-SHA`) that specifies the minimum cryptographic algorithm the Forge server/client will use during the SSL negotiation. If you omit this setting, the SSL software will try an internal list of ciphers, beginning with `AES256-SHA`. See the *Endeca Security Guide* for a list of cipher strings.

SSL-enabled script example

The following Control Interpreter script example enables SSL for the JCD, Dgraph (the MDEX Engine), and Forge.

In the example, global default settings are used to:

- Turn on SSL (with mutual authentication) for the JCD (via the `jcd_use_ssl` setting), Dgraph (`ene_use_ssl` setting), and Forge components (`forge_use_ssl` setting).
- Set the location of the certificate file for all the SSL-enabled components (`sslcertfile` setting).
- Set the location of the Certificate Authority file that all the SSL-enabled components will use to authenticate communications (`sslcafile` setting).
- Set the SSL cipher for all the SSL-enabled components (`sslcipher` setting).

```
##################### Global Variables ####################
#### Global variables can be reused anywhere in the control script, and
#### certain global variables (such as jcd_port and working_dir) are used
```

```
#### as defaults for the control interpreter.

# JCD connection
jcd_port = 8088
jcd_use_ssl  = true

# SSL settings
forge_use_ssl  = true
ene_use_ssl  = true
sslcertfile = $(endeca_root)\..\workspace\etc\eneCert.pem
sslcafile = $(endeca_root)\..\workspace\etc\eneCA.pem
sslcipher = DES-CBC3-SHA

# Other variables would go here, but are not shown in this example.

##################### Bricks#####################
#### Bricks define interfaces to various programs.
#### Endeca components such as forge, dgidx and dgraph have special bricks
#### that know about the process they are running.

# This brick defines the machine that will be used for the data update
process.
wine_indexer : Machine
 name = localhost

# This brick runs Forge to process the raw data. Note that SSL is
# used via the forge_use_ssl global setting.
wine_forge : Forge
 pipeline         = ../data/forge_input/Pipeline.epx
 forge_options    = -vw

# This brick runs the Dgraph. Note that SSL is turned on via the ene_use_ssl
# global setting.
wine_dgraph : Dgraph
 input       = $(wine_dgidx.output)
 port        = $(dgraph_port)

# Other bricks and scripts would go here, but are left out of this example.
```

Index

D

E

F

G

H

I

J

L

M

P

R

S

T

U