Chapter 5

NOI Part 4: Agent, AgentContext, International, Form, Name

This chapter discusses the classes relating to the Agent (but see Chapter 8 for a detailed discussion of how Agents are put together), AgentContext, International, Form and Name classes.

The lotus.notes.Agent Class

The Agent class represents the programmable aspects of Agents in Domino. You have the ability to locate, modify (to some extent), and execute Agent objects in a Database. See Chapter 1 for an overview of Agents and Chapter 8 for details on how they work.

You can locate Agent instances by using the Database.getAgents() and Database.getAgent calls.

Agent Properties

String getComment()

Returns the comment string associated with the Agent. A comment can be entered or modified using the Agent Builder UI.

String getOwner()

String getCommonOwner()

Returns the name of the last person to sign the Agent. Agents are signed when they are first saved and whenever they are modified. The getOwner() call returns a fully distinguished (hierarchical) name. GetCommonOwner() returns only the "common" part of the user's name.

lotus.notes.DateTime getLastRun()

Returns a DateTime instance whose value is the date and time the Agent last ran. If the Agent has never run, getLastRun() returns a *null*.

String getName()

Returns the name of the Agent.

lotus.notes.Database getParent()

Returns the Database object in which the Agent lives.

String getQuery()

If a search query was entered when the Agent was created, this call returns it. If there is no query, it returns *null*.

String getServerName()

void setServerName(String name)

The ServerName property designates the name of the server on which the Agent is allowed to run. This property is necessary because Agents are design elements, and as such they replicate along with the rest of the Database they reside in. If an Agent was by default allowed to run on any server, a replicated Agent would run on every server to which its Database was replicated, causing all kinds of conflicts and other grief. When an Agent is created, the ServerName property defaults to the current machine. You can edit it to any server name, but only one name is allowed.

As of Domino Release 4.6, you can enter an asterisk ("*") for the server name to indicate that the Agent can run on any server. Be very careful, though, you should be sure that the Database does not replicate to any other server, or you should be sure that you've coded the Agent in such a way that it will never cause replication conflicts. For example, if your Agent only creates new Documents in the Database, you should be okay. If your Agent modified existing Documents, you are definitely not okay.

If you modify the ServerName property you must invoke the Agent.save() method to commit your changes to disk.

boolean isEnabled()

void setEnabled(boolean flag)

Agents are enabled by default when they are created, if they are set up to run on a schedule (hourly, daily, etc.). The Enabled property has no meaning for Agents that are not scheduled. If you modify the Enabled property you must invoke the Agent.Save() method to commit your changes to disk.

boolean isPublic()

Returns *true* if the Agent is public, or "shared." If the Agent is not public, it can be seen or executed only by the person who created it.

Agent Methods

void remove()

Deletes the current Agent object from the Database.

void run()

Executes the Agent in the foreground. This is a synchronous call and will therefore "block" until the Agent is done. The Agent runs in the current process space, meaning that if you invoke this method from a program running on your workstation, the Agent code will be loaded into your machine's memory (regardless of where the Database containing the Agent lives) and run there. If you invoke this method from a background Agent running on a server, the Agent Manager process will load and execute the new Agent synchronously, suspending execution of the first Agent until the second is done.

Agents invoked this way on a client machine run with the privileges of the current user id. Agents invoked this way from background Agents run with the privileges of the signer, even if the signer of the second Agent is different from the first.

void save()

Saves the current Agent to disk. You must have Designer (or better) access to the Database, or this call will throw an exception.

The Agent is re-signed every time it is saved; therefore, this method will throw an exception if you invoke it from a server-based Agent (can't have just anyone signing Agents with the server's id, right?).

String toString()

Returns the name of the Agent.

The lotus.notes.AgentContext Class

As the name suggests, this class is only available to executing Agents. You get an instance of AgentContext from the Session class's getAgentContext() call (see Chapter 8 for details on how Agents are run by Domino). AgentContext is where all the information about the Agent's location and environment is accessed.

AgentContext Properties

lotus.notes.Agent getCurrentAgent()

Returns an instance of the current Agent object.

lotus.notes.Database getCurrentDatabase()

Returns an instance of the current Database, meaning the Database that the current Agent lives in.

lotus.notes.Document.getDocumentContext()

When an Agent is invoked via the Domino HTTP server, a "context document" is often provided, usually the submitted form. The context document is available to the Agent as an instance of the NOI Document class, allowing you to access all the data in the form. You can add to or modify this Document, but the HTTP server ignores any modifications.

If you're writing an API program that uses the C API to execute an Agent, you can supply, as the HTTP server does, an in-memory note handle to the Agent API to serve as the context document. If your Agent adds or modifies Items in the Document, your

API program could examine the Document after executing the Agent and make use of those changes.

String getEffectiveUserName()

When an Agent runs in the foreground (or in the background on a workstation), its privileges are those of the current user id. When an Agent runs in the background on a server, its privileges are those of the last signer of the Agent. One exception to this rule is that Agents invoked by the Domino HTTP server can run with the identity of the Web user instead of with the identity of the signer, if the Agent has been set up to do so (on the Agent Design properties box).

The EffectiveUserName property returns the name of the user under whose identity the Agent is running.

int getLastExitStatus()

Returns the status code stored with the Agent from the last time the Agent ran. A value of 0 means that the Agent executed without error.

lotus.notes.DateTime getLastRun()

Returns the date and time the Agent last ran (identical to the Agent.LastRun property).

lotus.notes.Document getSavedData()

When an Agent is created, Notes also creates a special Document to go along with it. The purpose of this Document is to allow Agents to store data persistently in the Database across Agent invocations. The "saved data" Document is a design element, not a data element, and so will never appear in any View. It is destroyed whenever its associated Agent is modified or deleted.

The advent of profile documents in Release 4.5 has by and large made use of the saved data document unnecessary. Access to the saved data document is not as efficient (for server Agents) as is access to profile documents, which are cached in the server's memory. Profile documents are saved forever, unlike saved data documents (which are deleted whenever the Agent is modified), making them a bit more generally useful. Saved data documents are also accessible only from their associated Agent (although you could have an Agent store the UNID of its saved data document somewhere, and then use that UNID later to access the saved data document directly), while profile documents are accessible from anywhere.

lotus.notes.DocumentCollection getUnprocessedDocuments()

The UnprocessedDocuments property is a DocumentCollection containing a set of Documents assembled at run time. The exact contents of the collection depend on how the Agent is configured:

- If the Agent is run from a View action button, and if the Agent is set up to run on "selected documents," then the collection contains the Documents that were selected in the View. You can access the current View by invoking Document.getParentView().
- If the Agent is scheduled and set up to run on "all documents that are new or modified since the Agent last ran," then the collection contains those Documents.
- If the Agent is set up to perform a search, the collection contains the results of the query.

For all other Agent configurations, this property will return an empty DocumentCollection.

AgentContext Methods

lotus.notes.DocumentCollection unprocessedFTSearch(String query, int maxdocs)

lotus.notes.DocumentCollection unprocessedFTSearch(String query, int maxdocs, int sortoptions, int otheroptions)

These calls are identical to the Database.FTSearch() calls, except that instead of operating on the entire Database, they operate only on the UnprocessedDocuments collection. They do not "refine" the UnprocessedDocuments collection by calling DocumentCollection.FTSearch(); instead they create a new collection instance.

lotus.notes.DocumentCollection unprocessedSearch(String query, lotus.notes.DateTime cutoff, int maxdocs)

This call is identical to the Database.Search() call, except that instead of operating on the entire Database, it operates only on the UnprocessedDocuments collection. It does not "refine" the UnprocessedDocuments collection by calling DocumentCollection.FTSearch(); instead it creates a new collection instance.

void updateProcessedDoc(lotus.notes.Document)

When an Agent is configured to operate on all Documents that are new or modified since the Agent last ran, the Agent gets a list of Documents called the *left to do list*. This list contains the set of Documents that the Agent has not yet processed. Agents set up this way must explicitly remove Documents from the left to do list; otherwise, those Documents will reappear in the list the next time the Agent runs.

The updateProcessedDoc() call does exactly that for a single Document. Another, often more convenient way to accomplish the same thing is to first get the UnprocessedDocuments collection, then invoke updateAll() on it. This is equivalent to invoking updateProcessedDoc() for every Document in the collection.

The lotus.notes.International Class

Like the AgentContext class, International provides contextual information, in this case about Domino's international settings. Some of the settings are specific to Domino, while others come from the operating system. The International class is composed of read-only properties—there are no methods.

International Properties

String getAMString()

String getPMString()

These strings are used for AM and PM in time formatting.

int getCurrencyDigits()

The number of decimal places used in number formatting.

String getCurrencySymbol()

The character or characters used to denote the local currency.

String getDateSep()

String getDecimalSep()

String getThousandsSep()

String getTimeSep()

The various characters used as separators in dates, times, and numbers.

int getTimeZone()

The current time zone. Might be positive or negative.

String getToday()

String getTomorrow()

String getYesterday()

Returns the strings used for special day names.

boolean isCurrencySpace()

If *true*, indicates that the local currency format uses a space between the currency symbol and the number.

boolean isCurrencySuffix()

If *true*, indicates that the currency symbol follows the number. Otherwise, the currency symbol precedes the number.

boolean isCurrencyZero()

If *true*, indicates that currency amounts between 0 and 1 should have a 0 preceding the decimal point. For example, $0.15, rather than $.15.

boolean isDateDMY()

boolean isDateMDY()

boolean isDateYMD()

These three properties indicate in what order the year, month, and day components of a date are displayed. Only one of these calls will return *true* in any session.

boolean isDST()

If *true*, indicates that the time format reflects daylight savings time.

boolean isTime24Hour

If *true*, indicates that the time format is a 24-hour format.

The lotus.notes.Form Class

The Form class allows somewhat limited access to the characteristics of a form. You can create a form instance (though NOI does not currently allow you to create forms programmatically) by using the Database.getForms() and Database.getForm() calls.

Form Properties

java.util.Vector getAliases()

As with Views, a Form can have both a name and one or more aliases. When you create or modify a Form you can specify additional names for it in the Design Properties box. The names are separated from each other by vertical bars. The first name in the list is the name of the Form; the others are aliases. Also as with Views, the names can have underscores in them.

This call returns a Vector containing a String instance for each alias of the Form. If the Form has no aliases, an empty Vector is returned.

java.util.Vector getFields()

Returns a list of field names used in the Form. The list comes from an Item attached to the Form named "$Fields." The contents of this Item are not always up to date, so don't assume that it is always accurate.

java.util.Vector getFormUsers()

void setFormUsers(java.util.Vector users)

java.util.Vector getReaders()

void setReaders(java.util.Vector)

The FormUsers and Readers properties let you control who gets to create Documents using the Form (FormUsers) and who gets to have read access to Documents created with this Form (Readers).

Each property is a list of user and/or group names. If a user is not in the FormUsers list (or in a group that is in the list), then that user will not be able to see the Form name in the Create menu, or otherwise be able to create a Document using the form.

The Readers property works a bit differently. When Documents are created using the Form, the Readers list from the Form gets copied to the Documents as the default $Readers Item. Users who are in the list (or in a group that is in the list) will have Read access to the Document. The Document creator can modify the Document's Readers list in the Document Properties box before saving or sending the Document.

Setting either of these properties causes your change to be committed to disk immediately.

boolean isProtectReaders()

void setProtectReaders(boolean flag)

boolean isProtectUsers()

boolean setProtectUsers(boolean flag)

Setting the ProtectReaders and/or ProtectUsers properties to *true* mark the Readers and/or FormUsers lists as being protected from deletion or modification by the

Replicator. Otherwise, it is possible that a new version of the Form will replicate into the database (or be brought in by the Design Refresh operation) and replace your Reader/User lists with its copy.

String getName()

Returns the name of the Form.

boolean isSubForm()

Returns *true* if the Form is a subform.

Form Methods

void remove()

Deletes the Form from the Database.

String toString()

Returns the name of the Form.

The lotus.notes.Name Class

The Name class is a nice little utility for parsing distinguished names. You create a Name object using the Session.createName() call, passing a String in as the argument. If the String is a hierarchical name, the Name properties will return various pieces of that name.

A fully distinguished name includes keyword tags designating special parts of the name as meaningful. For example the distinguished name CN=Bob Balaban/O=Looseleaf has two tags in it: the CN= part designates Bob Balaban as the common name, and the O= part designates Looseleaf as the organization. Lots of additional tags are available. For each of the Name properties, I've also provided the distinguished name tag that goes with it.

The Name class has no methods.

Name Properties

String getAbbreviated()

Returns the abbreviated form of the distinguished name. For example, CN=Bob Balaban/O=Looseleaf is returned as Bob Balaban/Looseleaf. To go from an abbreviated form to a fully distinguished form, use the getCanonical() call.

String getADMD()

Returns the administration management domain name associated with the user name. The tag is A=. If there was no A= tag in the original name, this property returns *null*.

String getCanonical()

Returns the canonical (unabbreviated) form of the name.

String getCommon()

Returns the common part of the distinguished name. The tag is CN=.

String getCountry()

Returns the country part of the name. The tag is C=. If there was no C= tag in the original name, this property returns *null*.

String getGeneration()

Returns the generation part of the name, such as Jr. The tag is Q=. If there was no Q= tag in the original name, this property returns *null*.

String getGiven()

Returns the given name. The tag is G=. If there was no G= tag in the original name, this property returns *null*. There is no reliable way for NOI to parse a given name out of a common name, especially if you consider the international implications. Europeans are used to having a person's given name come first, but in many Asian languages the family name comes first. In the name Yip Wai-ki, for example, the given name is Wai-ki, the surname is Yip.

String getInitials()

Returns the initials belonging to the name. The tag is I=. If there was no I= tag in the original name, this property returns *null*.

String getKeyword()

Returns the part of the hierarchical name known as the *keyword*. The keyword consists of the following pieces of the name, if present, with backslash separators: country\organization\organizational unit 1\.organizational unit 2\.organizational unit 3\.organizational unit 4.

String getOrganization()

The organization, usually the company name. The tag is O=. If there was no O= tag in the original name, this property returns *null*.

String getOrgUnit1()

String getOrgUnit2()

String getOrgUnit3()

String getOrgUnit4()

Returns the specified organizational unit component. An organizational unit is usually a division, department or location identifier within an organization. Lotus, for example, uses office location as an organizational unit in its employees' Notes ids. The tag used for all organizational units is OU=, and you can have up to four of them in a name.

If there was no OU= tag in the original name, these properties return *null*.

String getPRMD()

Returns the Private Management Domain part of the name. The tag is P=. If there was no P= tag in the original name, this property returns *null*.

String getSurname()

The surname, or family name. The tag is S=. If there was no S= tag in the original name, this property returns *null*. There is no way for NOI to reliably parse the surname out of the common name.

boolean isHierarchical()

Returns *true* if the Name is hierarchical.

String toString()

Returns the canonical name.

Summary

Next, Chapter 6 concludes our in-depth discussion of NOI with the Registration, Newsletter, and Log classes.