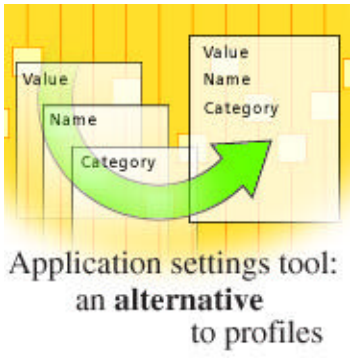


by
Jonathan
Coombs

Level: Intermediate
Works with: Designer 5.0
Updated: 07/02/2001

In the process of developing an application, you nearly always identify certain system features that need to have configurable functionality. For example, the application's owner may need to be able to modify the list of options displayed in a combo-box on a form. Or the text of all system-generated e-mails may need to be periodically updated to reflect changes in corporate policy.

In some Notes environments, the developer of each application is responsible for making changes directly to the production databases. In such environments, you might be able to change an application's configuration manually by making minor design changes. However, most environments—and good development practices in general—dictate that the production design should not be modified frivolously. In a typical environment, you first complete a new release of the application on a development server and then formally request that the Notes administrators move it into production.

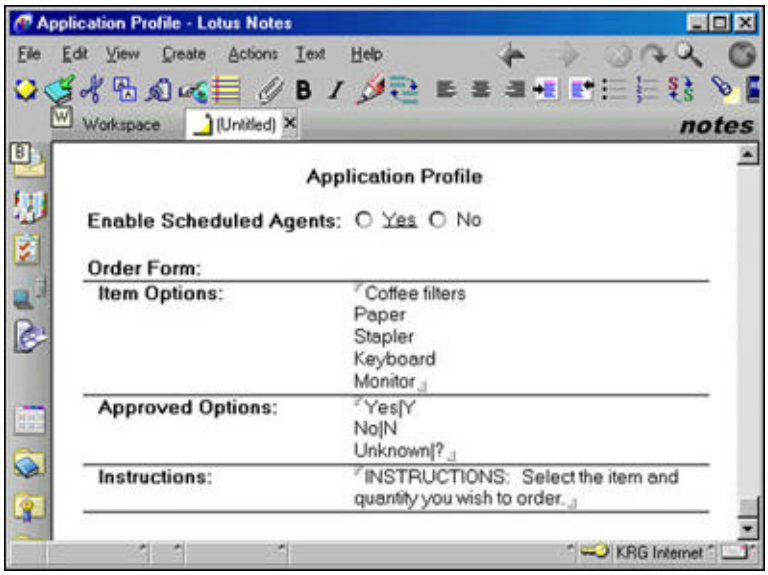
In general, then, design changes should be used only for bug fixes and true enhancements, and configurable settings should be modifiable through a "system configuration" or "application settings" interface. (Such features are also referred to as control, keyword, or profile documents.) If this interface can be made user-friendly and robust, you can give the application owners the responsibility of maintaining some of these settings. If this interface can also be made generic and reusable, it can increase the system's flexibility and significantly reduce development time.

This article presents a reusable Application Settings tool in the context of a fictitious order-tracking application and covers a few important development standards that increase the tool's reusability. You can download and examine the sample **Simple Settings database** discussed in this article from the **Iris Sandbox**. Since the Application Settings tool will support both the Notes and Web clients, the article also covers some basic techniques for maintaining compatibility with both.

This article assumes an intermediate understanding of designing Notes/Domino applications.

Design alternatives

The most typical example of an application settings interface is a single profile document containing one field per setting. The instructions and data type for each field are hard-coded into the profile form's design. For example:



This approach has the advantage of being simple and centralized, and of leveraging the Notes functions that provide quick access to profile documents, such as @GetProfileField and NotesDatabase.GetProfileDocument.

There are disadvantages, however, to using profile documents. The main disadvantage is that they are hidden, making them difficult to find, delete, or copy between databases (for example, from development into production). It is also difficult to build an interface for viewing and maintaining multiple profile documents. If all of an application's settings have to be stored in a single document, it is hard to organize all of those settings meaningfully on the document's form.

Because of these problems, I usually avoid using profile documents. Instead, I store application settings in ordinary Notes documents tucked away in an administrative view, using a generic form to create a separate document for each setting.

This approach does have its disadvantages as well. It adds an additional design element (the view) and uses indirect lookup functions (@DbLookup and NotesView.GetDocumentByKey), which can negatively impact performance in some cases. Even so, the ability to create and maintain any number of settings is necessary when building a reusable Application Settings tool, so I think this flexibility outweighs the disadvantages.

So what does a generic Setting form look like? The Setting form presented in this article is very simple in concept. Since each setting is stored as a separate document, the form essentially only needs two fields: a text field to store the setting's unique name and a text field to store its value. But why stop there? By adding a field for instructions, each setting can be made to display a message explaining its purpose and use. Adding an Authors field gives us the ability to assign specific settings to specific administrators. Adding different types of value fields can make the user interface support various data types in addition to simple text.

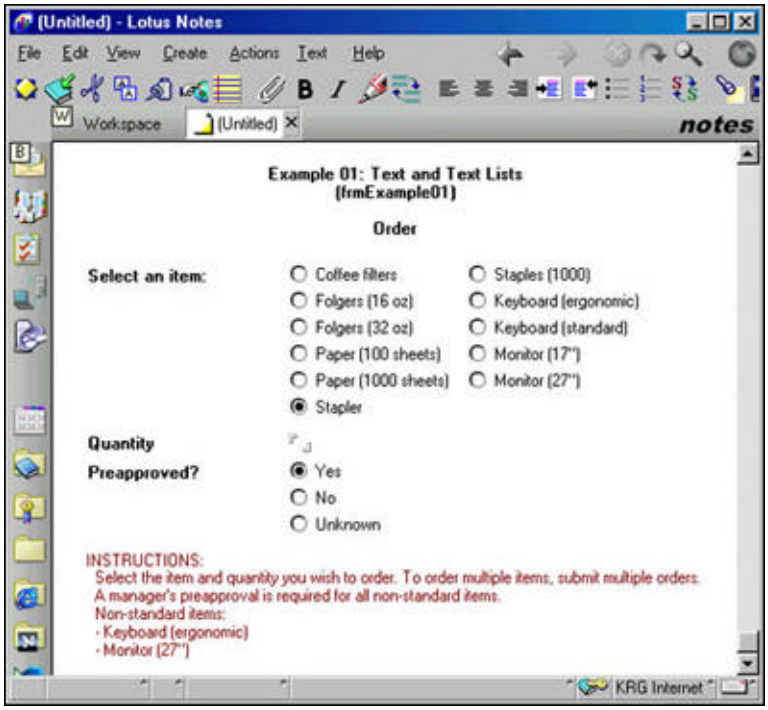
Notice that this approach is not tied to the design of any particular application. It puts no restrictions on the number of settings an application can have; the application could be designed to use a variable number of settings, or even hierarchies of settings. Once the tool has been fully developed and debugged on all clients, it can be used in a wide variety of

applications without modifying its design. If element-level design inheritance is enabled, bug fixes and enhancements can be deployed from a central template. (As with any reusable tool, delivery and maintenance should be carefully planned. If you choose not to use design inheritance, beware of creating a different flavor of the tool for each application. See the **Domino 5 Designer Help** for more on design inheritance.)

A basic implementation

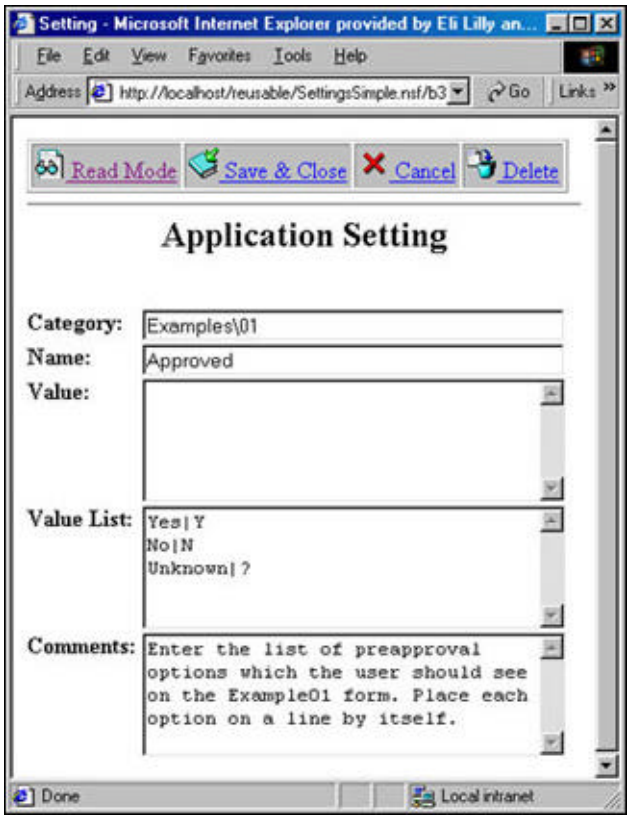
Let's start by looking at a basic implementation of the Application Settings tool that is fully functional from both Notes and Web clients and that can be easily upgraded to a more advanced tool. The Application Settings tool consists of a form, a user view, and a hidden lookup view. These elements, along with an Order form and view that demonstrate their use, are included in the **Simple Settings database** (SettingsSimple.nsf) available in the **Iris Sandbox**.

Before we examine how the Setting form and views of the Application Settings tool itself are put together, here's an overview of how the tool works within the sample order-tracking application. The sample Order form, below, demonstrates the use of text and text list settings.



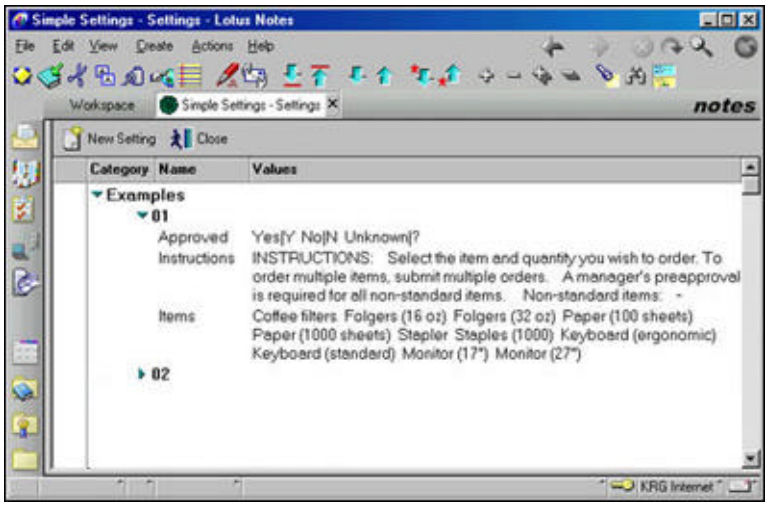
Although the instructions and radio button options could be built into this form, the form will be more flexible if the application owners can control them using Setting documents.

To store this Order form's instructions and option lists, the Application Settings tool will need to store one text setting and two text list settings. Each setting will be individually created with the Setting form and stored in its own document. For example, the screen below shows a Setting document for one of the text lists, the Approved field. (The Category field on this form is there to help keep large numbers of settings organized.)

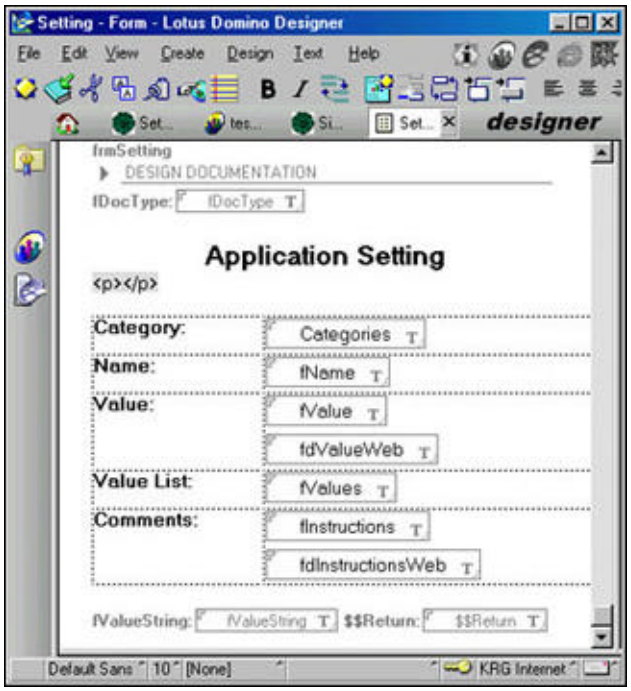


This basic version of the Setting form is easy to implement and makes application settings available to both Notes clients and, in this case, Web clients. Having a Web interface as well as a Notes interface may not be necessary if your application administrators will access the application only with Notes clients; but since we are developing a tool that can be used in many different applications, it's a good idea for the tool to support both clients.

As the Setting documents are created, they appear in the Settings view. The screen below shows the Settings view with the Setting documents for the Order form's settings. The Order form's fields can then use @DBLookup commands to retrieve the settings from the Setting documents. The syntax of these lookup commands is described later in this article.



Setting form design standards
Before explaining the design of the Setting form, let's take a look at the naming and design conventions it follows.



Near the top of the form is a collapsed section containing embedded design documentation. I've included documentation in the form itself because developers are most likely to see it there and because it goes wherever the form goes. (Some additional documentation is provided through REM statements in individual field formulas.)

Each field name is prefixed with an *f*, and also with a *d* if it is a display field. This helps to distinguish it from the field names reserved by Notes for special purposes (for example, Form, Server_Name, and SendTo). Hidden fields are small (8 point) and gray, and usually have bold labels next to them. (When unhiding several hidden fields for debugging purposes, the labels make it easy to tell which is which.)

Another important consideration with any Web editing form is page caching. Most users' browsers store recently accessed pages on the local hard drive. This improves performance when browsing static pages, but it can also prevent those users from seeing their most recent changes if they edit, save, and reopen a document. To tell the browser to always request a fresh page, you can set the form to "Automatically enable Edit Mode," or use an HTML <META> tag (in the form's HTML Head Content formula) to give the page a past expiration date:

```
"<META HTTP-EQUIV=\"Expires\" CONTENT=\"Mon, 06 Jan 1990
00:00:01 GMT\"></META>"
```

The Setting form

The basic version of the Setting form allows anyone with Editor access to create, name, and categorize settings that can be stored as text or text list values. It also provides an Instructions field for storing an explanation along with each setting. Let's look more closely at each of these fields (as shown in the figure above).

The hidden fDocType field always computes to "Setting" and is used by the Settings views' selection formulas. (Using a custom field yields better long-term flexibility than using the Form field.)

The Categories field does not have the *f* prefix because its name has special meaning to Notes. Users can categorize individual Setting documents by typing a category title into this field, or (in Notes) they can categorize several documents at once by selecting Categorize from the Actions menu. (Note that Categories fields should be set to allow multiple values, since the Categorize feature returns a text list.)

The fName, fValue, and fInstructions fields are all single-value text fields, so Domino automatically sends them to the Web client using HTML <INPUT> tags. This is adequate for the short fName field, but the fValue and fInstructions fields need to support multiple lines of text. So, the fValue and fInstructions fields are only sent to the Web client when in read mode. When editing from the Web, the fdValueWeb and fdInstructionsWeb display fields are used instead. Their formulas generate HTML <TEXTAREA> fields named fValue and fInstructions, respectively, so these text areas' values are saved back into the proper Notes fields on submit.

There are a few things to watch out for when using this technique. It cannot be used if the form's "Generate HTML for all fields" property is enabled, as this will generate duplicate field names on the page. Also, the newline characters returned by the browser may not be interpreted correctly by Notes. The ASCII characters 13 and 10 are used to indicate carriage returns and line feeds, but they are not always used consistently. In the browsers I've tested, a 13 followed by a 10 is equivalent to one @NewLine in Notes, and then any individual 13 or 10 is also equivalent to one @NewLine. Therefore, I added these translation formulas to the fValue and fInstructions fields, respectively:

```
@ReplaceSubstring (fValue;
(@Char(13)+@Char(10)):@Char(13):@Char(10); @NewLine)
```

```
@ReplaceSubstring (fInstructions;
(@Char(13)+@Char(10)):@Char(13):@Char(10); @NewLine)
```

The next field, fValues, is a multi-valued text field set to display multiple values on separate lines, so Domino automatically generates a <TEXTAREA> field for it and handles any newline characters

appropriately. This field's text area, therefore, requires no special display field or translation formula.

The hidden fValueString field stores each setting's value in an alternate standard format. It first determines whether the setting's value was entered into fValue or fValues, and then converts the value into a standard string:

```
@If (fValue != ""; fValue; @Implode(fValues; @NewLine))
```

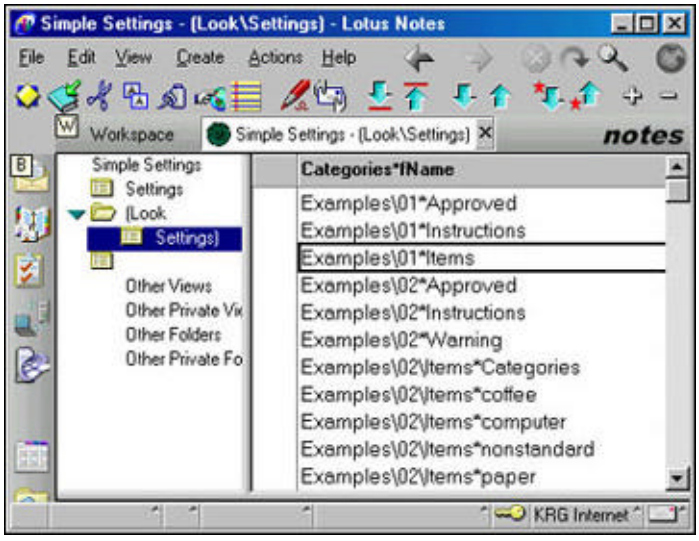
Finally, the hidden $$Return field tells Domino to display the Settings view after the Web form has been submitted:

```
path := @ReplaceSubstring (@Subset(@DbName;-1); "\\"; "/");
"[/" + path + "/" + "vwSettings" + "]"
```

In addition to all these fields, the Setting form includes a standard set of actions: Edit Mode, Read Mode, Save & Close, Cancel, and Delete. To make the form layout look a little better from the Web, I also added a <P> tag below the title as pass-thru HTML and used the fields' HTML attributes to set their widths. You may wish to make other cosmetic changes as well, but always keep in mind that the form should be as generic and reusable as possible.

The views

The views (vwSettings and vwLookSettings) provide access to Setting documents for viewing, editing, and code lookups. This functionality could be combined into a single view, but instead I have created one user interface view, the **Settings view** shown above, for viewing/editing and a hidden view for code lookups, shown below. This is an extremely important design standard because it allows you to freely rearrange your user interface views to fit your users' needs without breaking any code lookups.



The views are documented with REM statements in their selection formulas, but let's take a quick look at them.

As mentioned before, both views select just those documents whose fDocType field equals "Settings." The **Settings view** is a formatted, categorized view that contains actions the user can select, while the Lookup view is completely unformatted and displays only a unique key for

each document. It is designed to support LotusScript lookups and formula lookups by field name. It does not have columns for fValue or fValues because lookups made against column numbers tend to make the view design less flexible (and the lookup code less readable). Still, if you find your application requires faster lookups, it might be necessary to add lookup columns.

Since each setting is uniquely identified by the combination of its category and name, the lookup view defines its lookup key as the concatenation of the two, separated by an asterisk (*). The Order form runs three lookups against this hidden view, one for its Instructions field and one for each of its radio button fields. Since the Instructions field needs the single text value stored in the "Examples\01\Instructions" setting, its key is "Examples\\01*Instructions" and it accesses the fValue field:

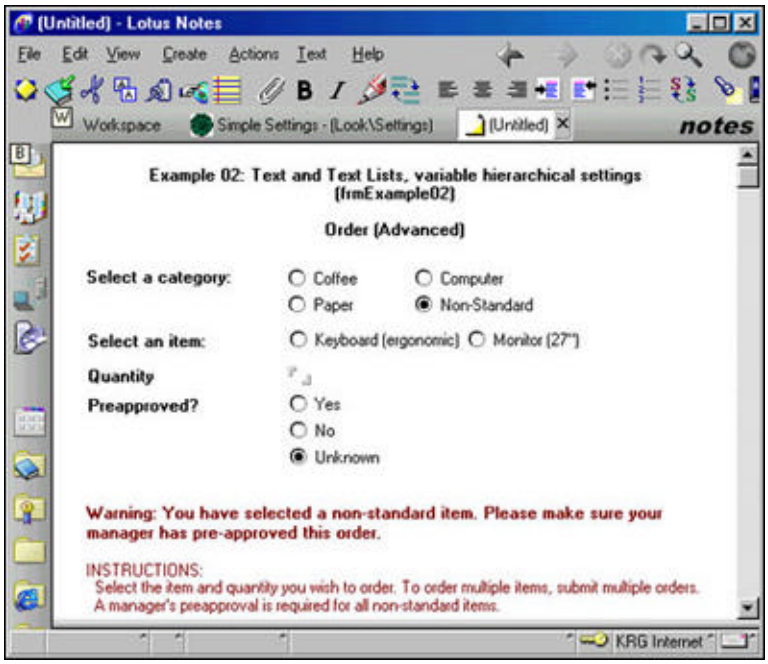
```
key := "Examples\\01*Instructions";
temp := @DbLookup("Notes":"NoCache"; ""; "vwLookSettings"; key;
"fValue");
@If( @IsError(temp); "Error: Unable to find the Setting document (" + key
+ ")"; temp)
```

The lookups for the two radio buttons' options are identical, except that they have their own unique keys and retrieve their values from the fValues field.

Creating dynamic setting structures

To demonstrate the flexibility gained by using a generic form and one document per setting, the sample database also includes an Advanced Order form. This form supports a variable number of settings stored in a simple hierarchy.

The original Order form displayed all of its items in one long list. The Advanced Order form provides the same items, but it breaks them down into categories.

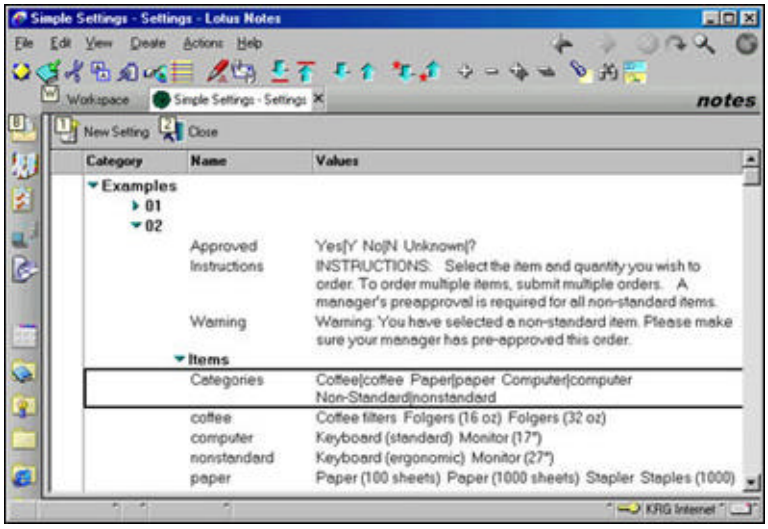


The list of categories is maintained in the "Examples\02\Items\Categories"

setting and displayed by the category field. Whenever its value changes, this field refreshes the form, causing the item selection field to display the list of items that correspond to that category. The lookup that returns this list is very similar to the one used in the original Order form, except that it uses the current category as part of its lookup key:

```
key := "Examples\\02\\Items*" + fItemCategory;
temp := @DbLookup("Notes":"NoCache"; ""; "vwLookSettings"; key;
"fValues");
@If( @IsError(temp); "Error: Unable to find the Setting document (" + key
+ ")"; temp)
```

To support this dynamic association of categories and items, I broke the original Setting document ("Examples\01\Items") into multiple documents. There are currently four categories, but the Order form will support any number of categories without a design change, as long as each one corresponds to a value in the "Examples\02\Items\Categories" setting. Each category listed on the Advanced Order form has a corresponding Setting document, as seen in the Settings view below.



Possible enhancements

This article has presented the basic structure of a reusable Application Settings tool and a few examples of how it can be used with Notes and Domino to support individual settings, a variable number of settings, or a structured hierarchy of settings. The user interface for this tool is not sophisticated, but it is usually adequate for applications that require little maintenance and have well-trained administrators.

The most significant problem with the basic Setting form is its simple interface. It does not directly support non-text data types, so dates, names, and numbers must all be entered as unvalidated strings. It is also somewhat confusing because it does not distinguish between developers and application administrators. That is, everyone with Editor access has full access to every setting, even though in most cases, the administrator should only edit the Value or Value List field (but not both). A design that gave full access only to developers would be safer. You can address these and many other issues by enhancing the Application Settings tool.

These and many other issues can be addressed by enhancing the Application Settings tool. I have begun this process by creating an Advanced Setting form that supports multiple data types, applies user

roles, and improves the user interface. It also provides a script library to simplify LotusScript lookups. This enhanced version of the Application Settings tool, **Advanced Settings sample database**, is available from the **Iris Sandbox**.

About Jonathan Coombs
Jonathan is a software developer for **Joseph Graves Associates, Inc.** in Indianapolis. JGA is a full service consulting firm that delivers quality IT services and customized e-commerce, Internet, and document management software solutions. Jonathan's professional interests include software reuse, Lotus Notes and Domino technology, and computational linguistics. He can be reached at **jonathancoombs@bigfoot.com**.

