"Whose Rule Is It Anyway?" – A Case Study in the Internationalization of User-Configurable Business Rules

Morgan McCollough

Bridge360
1016 La Posada Dr, Suite 120
Austin, TX 78752
morgan_mccollough@bridge360.com

Abstract. This paper consists of a case study concerning the internationalization of an electronic invoice management web application and its central rules engine. It examines the challenges faced in introducing internationalization changes at the level of a custom scripting language processor and the problems inherent in maintaining compatibility with existing deployments. The paper outlines the specific solution and the ways in which the key concepts of locale context and lazy initialization may be applied to other similar internationalization problems.

1 Introduction

In early 2007 Bridge360 was challenged with the internationalization (i18n) of a client's flagship web application. The client was taking the first steps into the European market and had limited experience with global software. The application in question was a web-based, ASP.NET system developed for large organizations to automate the process of managing electronic invoices. The system was originally an outgrowth of an effort to establish standards for certain types of electronic invoices and tools to handle them. The invoice management system created value for large organizations by requiring the electronic submission of all invoices using a specific standard and serving as a conduit for invoice approval workflows and integration with accounts payable.

The particular challenge concerning this case study is not the internationalization of the invoice management application in general, but rather one of its core components, the rules engine. The rules engine is a custom scripting language that provides a framework for customers to develop and add their own business processes, rules, and workflows to an existing system. For example, a billing management system might be customized through the use of a rules engine to automatically apply a certain percentage discount for all clients in a particular region. In the context of electronic invoices, rules could be developed to automatically apply rate adjustments to invoices arriving from particular vendors, to raise errors when an invoice is submitted against a project not approved for payment, or to insert warnings directly into an invoice when the total amount billed for a particular project climbs over a certain threshold. In general the strength of a rules engine lies in the ability to customize the behavior of a software system for any particular client in the field.

N. Aykin (Ed.): Internationalization, Design, LNCS 5623, pp. 274 – 282, 2009.

The rules engine in this specific application was implemented using a custom scripting language processor, which allowed for a great deal of flexibility in customization for customer deployments due to its expressive power. However, this flexibility also presented complications in the area of internationalization. This case study will focus on the challenges encountered when implementing a solution capable of supporting global environments. The difficulties in the project centered on two central issues. First, the server had to run in one language environment but service users in different languages. Second, the locale under which the rule was initially executed did not necessarily match the locale under which its results were viewed. The final solution will also be described, along with the core concepts of locale context and lazy initialization, which may be applied to other similar internationalization problems.

2 The Project

The first step in the project was an internationalization assessment of the automated invoice management system. This is a useful technique to gain an overall picture of an application's code base and discern its internationalization shortcomings. Despite the fact that the server application was written in C#.NET, a language with a large amount of internationalization support built-in, the assessment revealed a number of potential problems. This was largely because the system was never architected to support multiple languages and hence included a number of U.S.-specific assumptions as well as places where the built-in i18n support of C# was not utilized. The application also employed a number of 3rd party components that were incapable of performing correctly in an international environment.

2.1 Rules Engine

The rules engine in the client's invoice management system was implemented as a custom scripting language processor. In other words, the client created a completely new, custom scripting language, solely for the purpose of defining business rules to modify the behavior of the system. The rules engine consisted of a single process which ran the language interpreter to parse, compile, and execute the various rules scripts defined in the engine's configuration file. As invoices were posted into the system, rules were executed in the context of each invoice as a whole or for each individual line item in an invoice.

The scripting language was straightforward but highly configurable. It was never intended to be a fully-expressive programming language and therefore consisted mainly of conditional statements, a few basic mathematical operations, and a small set of actions or functions that could be performed on an invoice. Each defined rule consisted of a context definition (invoice or line item), a conditional expression to examine the incoming invoice data and make a logical choice of whether or not to execute the rule, and an action to perform if the rule in question was triggered. Support for conditional statements included basic logical operators as well as access to a predefined data structure consisting of all of the main elements of an invoice. Actions were a series of functions with specific parameters as defined in an external configuration file. This configuration file defined the names of all supported actions, their

parameters, and references to the code that implemented each action. It was therefore relatively simple for the client's professional services department to add custom actions during a product deployment if the default actions did not meet a customer's requirements. In addition, there was a completely separate client application developed for editing business rules in the field. It provided a graphical environment for the Professional Services group to quickly author a series of rules to meet any customer's needs.

An Example Rule. The following is an example of a simple rule that could have been implemented in the invoice management system's rules engine.

```
If
          Invoice.Project.ID == "ACCT-257”
           and
          Invoice.Total > 100000.00
Then
          AddInvoiceWarning “Invoice submitted on “ +
            Invoice.SubmitDate + “ exceeds the maximum
billable amount
            for Project “ + Invoice.Project.ID + “(“ +
100000.00 +
            “) and will require special approval!”
End
```

The above rule would have the effect of adding an invoice warning to an incoming invoice if the ID of the related project was "ACCT-257" and the total billed in the invoice exceeded 100,000. An invoice warning is a special message that is attached to an invoice that is prominently visible at the top of the invoice detail screen in the application's web interface. It would be one of the first messages displayed when a user viewed an invoice matching the above condition through the web interface. The message itself contains several variable expressions that are part of the pre-defined data structure mentioned above. They provide access to the various fields and values contained in an invoice and resolve to the appropriate values when the rule executed.

2.2 The Problem

The difficulties in internationalizing a rules engine like the one described above are twofold. First, all existing rules had to function correctly when the application was deployed on a non-English application server. Many non-internationalized applications exhibit problems or cease to function at all when run on a non-English operating system. Even if a non-U.S. customer were to deploy the English version of the application, it is likely that the application would be run on a non-English version of the Windows server platform. Requiring a customer to run an international software product exclusively on an English operating system is not a reasonable option. Second, as evidenced by the example above, this particular rules engine was capable of

adding strings to an invoice that were visible in the main application interface. Therefore there had to be some mechanism to translate this text since the application was a web-based system where a single server could support users in multiple languages simultaneously.

Constraints. Although the basic problem is outlined above, there were a number of important additional requirements to consider in the specific situation of the automated invoice management system. These requirements or "constraints" had to be taken into account when designing and implementing an appropriate solution.

Backwards Compatibility. The internationalization improvements were being incorporated in the main version of the application as opposed to an international-specific version. The client had a significant base of existing customers that all had rules configured specifically for their business environments, and there were plans to upgrade a number of these customers to the next major release, which would include the internationalization features. If *any* changes were made that caused existing rules to stop functioning after an upgrade or required changes to existing rules the costs involved would have been unacceptable to current customers.

Multi-language Server. The client confirmed that their first European customer would be hosting the application to run in both English and French simultaneously. There are a number of companies, especially in Europe, that do business in multiple languages, and requiring clients to install a separate server for every desired language would have been unreasonable due to the increased costs of using multiple servers and the lack of an easy way to share data among multiple servers without significant application enhancements. This type of web application has further complications because the user interface locale is variable and does not always match the locale of the server that is running the application. This presented a challenge for the rules engine since it ran in a separate process from the user interface. The rules processor ran using the locale of the server, which could be different from the locale of a particular user session in the web interface. However, the user's locale defined the environment in which a rule's output would be viewed, including examples like the one above.

Dynamic Data. In the example above there are variables inserted into the middle of the invoice warning string. These types of variables are sometimes referred to as "dynamic data" because their value is not determined until run-time and hence may change depending on the specific execution environment. In the example, "`Invoice.SubmitDate`" would be replaced with the date on which the invoice in question was submitted, and this would change depending on the date on which the rule runs. The rules scripting language supported data insertion for any data field in an invoice, which meant that inserted values could be strings, numeric values, currency values, or dates. These values therefore needed to be rendered according to the user's locale when viewed in the user interface. Deciding upon a locale context to use presented a problem because rules processor executed completely independently of the user interface.

Customization in the Field. The final limiting factor concerned the nature of the rules engine itself. The rules engine was meant as a customization tool, and the client configured it mostly during deployments at customer sites. It was therefore logical that

there would need to be a method to create internationalized rules in the field. This spurred the need for a mechanism by which the Professional Services, or even the customer, could create and configure fully internationalized rules and update any necessary translations.

3 The Solution

The final solution to the internationalization problems in the rules engine came down to solving 3 major issues. First, the application had to correctly execute the custom script rules regardless of the language and locale of the server's operating system. Second, dynamic data had to be rendered correctly at runtime depending on the user's locale. Finally, there needed to be some method to translate arbitrary strings utilized in custom business rules without breaking any existing syntax or rules. The following sections outline how each of these problems was solved and how each solution addressed the requirements and constraints listed above.

3.1 Locale Context

When writing code for an internationalized application, it is important to be cognizant of locale. Any piece of code that executes on modern operating systems does so in a particular locale context, and there are many libraries and system functions that will change behavior depending on the locale environment. This is especially true for a language environment like C#.NET where all string-related operations utilize the configured locale context to define their behavior. Many internationalization problems can be traced to code that completely ignores locale or simply assumes behavior based on the rules and conventions of one particular locale.

In the case of the invoice management rules engine, the scripting language processor was largely a string parser and interpreter that assumed the conventions of the U.S. English locale. When this code was executed on a non-English machine there were a number of issues. All string-related functions in the C#.NET libraries automatically picked up the locale of the system, which broke many implicit assumptions in the code. For example, the code assumed a period was always used for the decimal point. However, the French locale uses a comma instead of a period which caused the script language processor to parse many numbers incorrectly.

Two basic solutions to this problem were considered. The first was to make sure the locale was properly detected and then go through the code and eliminate any assumptions based on U.S. conventions. This approach had two major problems. First was that the rules engine was at its core a scripting language interpreter and hence had many lines of code. Combing through this component to find all the potential problems would have been a long process, especially considering project time constraints and the fact that there were few people left at the client that were familiar with the interpreter's inner workings. The second issue was that there was a significant base of default rules that shipped with the product and were used as the basis to customize a client's system. Forcing the interpreter to use the system locale could have potentially required the client to keep different versions of these scripts for each language that differed only in syntax, due mainly to differences in date and numeric literal values.

The second solution, which was the one chosen, was relatively simple. Code was added in a number of places to explicitly set the locale environment of parse operations to use U.S. conventions. This was to ensure consistency such that any rules script running in any language environment would be functionally identical. The rules scripts themselves were never seen by end users and it would have made no sense for the same script to operate differently in two different language contexts or to require rules to be re-written depending on the language environment of the server, as it was assumed that international users would no longer embed literal strings into business rules. This also made sense because the rules script was essentially a programming language, and programming languages tend to follow the U.S. conventions as a standard. Finally, explicitly using the U.S. conventions was a low-risk change because the engine had already been thoroughly tested on an English operating system.

3.2 Lazy Initialization

Lazy initialization is a concept in computer science whereby the creation of an object or the calculation of a value is delayed until such time as it is actually needed versus performing the operation ahead of time and storing the result for later use. The same basic principle was applied to the evaluation of data in the rules engine. Originally, all expressions in a rule were evaluated at the time of the rule's execution. In the example rule above, the invoice warning string would have been completely evaluated and attached to the invoice in its final form, e.g. “Invoice submitted on 4/5/09 exceeds the maximum billable amount for project ACC-257 (100,000.00) and will require special approval!” This was no longer possible in the context of a multi-language server. The locale under which the rule was executed did not necessarily match the locale of the user that logged into the web interface to view that specific invoice. In the case of a user session in French, the date listed above would not match any of the other dates displayed in the internationalized interface and it would therefore be easy for the user to mistake it for May 4, 2009 rather than April 5, 2009. It also would have made translation of the string at run-time for different user sessions extremely difficult.

The output string resulting from rule execution was stored in a table in the application’s database. When the corresponding invoice was displayed in the user interface, all of the various comments, warnings and other strings were pulled from the database and displayed directly in the UI. In order to make the system capable of displaying multiple language versions of these strings, there had to be some locale-neutral version of the evaluated string that could be transformed into the correct language form at run-time based on the user’s locale. This concept of lazy initialization is the basic idea behind the internationalization of user interface strings for most global software. The differences lie in the exact mechanism used to achieve it.

Several alternatives were considered regarding where and how to store the translations for these user-defined strings. The first idea was to have the rule authors simply place multiple translations of a string in the rule script itself. This was quickly dismissed as a maintenance nightmare, as it would require editing rule scripts every time a language was added and would also require significant modifications to the scripting language itself. It would also not solve the problem of dynamic data. The second idea was to assume users would only ever look at invoices related to specific projects and all users under a specific project were likely to speak one language. If that were

the case, there could be a specific language designated for a specific project, and the rules could be segregated by language and project. That again was quickly dismissed, as it assumed too much about the way in which the application would be used. It also did not solve the problem of dynamic data, as there was no way to change the locale of the rule execution and hence the format of numbers and dates at a project level.

Ultimately, the solution was to internationalize the rule code in much the same was as normal application code. Instead of embedding a literal message in rule code, a string identifier would take its place. For example a message like “Invoice exceeds maximum billable amount" would be replaced by an identifier that could be used to look up the actual value when a user viewed the invoice through the web interface. Rule output with any identifiers would be inserted in the database table when the rule executed, just as if it were a typical string. A hook was added to the user interface layer to tell these identifiers apart from normal text when rule output was pulled from the database. When the user interface detected an identifier within rule output, the real value would be pulled from a rules engine resource file according to the user’s session locale. Any dynamic data was inserted into the translated string properly formatted for the user’s locale before being displayed in the web interface. In this way, final evaluation of rules data could be delayed until the appropriate locale could be obtained. The main problem with this approach was that it required a consistent format to be used for the special string identifiers that for the sake of backwards compatibility could not be confused with normal text data. Otherwise, there was a very real danger of breaking existing scripts. This, and the necessity to support dynamic data, led directly to the final piece of the solution, the introduction of custom syntax into the scripting language processor.

3.3 Custom Syntax

As mentioned above, the introduction of new syntax into the rules scripting language was required to produce a consistent pattern that could be evaluated correctly at the user interface level without changing the behavior of any of the existing syntax.

Originally, following the convention of lazy initialization, it was thought that for dynamic data variables, the simplest solution would be to store the variable name itself and delay evaluation until the user interface layer. For example, store the variable “`Invoice.Total`” as a string with a marker to identify it as a variable. However, it quickly became clear that delaying evaluation for all variables used in message strings could greatly complicate the code in the user interface layer. It was also realized that there were many situations where the value of the variable could change between the time of the rule’s evaluation and when the data was eventually displayed in a web page. For example, if the variable “`Invoice.Total`” was saved as an identifier and not resolved to its numeric value until displayed to the user, its value might change if any users added discounts or rate adjustments to the invoice after it was posted into the system. In the example rule given above, this would cause a problem if the adjustment lowered the invoice total below the threshold that triggered the rule in the first place since the warning comment would no longer make any sense.

Eventually, the team decided that the dynamic variables would have to be evaluated at the time of rule execution. However, in cases where they were to be used in translatable strings, they had to be evaluated into a locale-neutral format that could be

re-interpreted later in order to format them properly for display in the user interface. This forced the creation of a special syntax to avoid impacting the normal evaluation behavior of dynamic data and therefore run the risk of breaking existing rules.

First, a new string concatenation operator (a comma) was added to the language to serve as a companion to the existing string concatenation operator ("+" in the example above). The use of the new operator allowed for the conversion of date and numeric data into a consistent locale-neutral format. Second, a convention was established and documented that any string value in a rules script starting and ending with 2 dollar signs ("$$") would be interpreted as a literal string by the rules processor but would be interpreted as a key to look up a translated value by the user interface lazy initialization hook. Values to be inserted into the string at run-time could simply be concatenated behind the key value using the new concatenation operator.

The example rule given earlier would be rewritten as follows using the new conventions and syntax.

```
If
          Invoice.Project.ID == "ACCT-257"
           and
          Invoice.Total > 100000.00
Then
          AddInvoiceWarning "$$AmountExceededWarning$$",
           Invoice.SubmitDate, Invoice.Matter.ID,
100000.00
End
```

When evaluated by the newly modified rules engine, the above rule would generate the follow warning string to attach to an invoice.

```
"$$AmountExceededWarning$$;2009-4-5;ACCT-257;100000.00"
```

The actual warning text was placed in the rules engine resource file using the given id as follows.

```
AmountExceededWarning = "Invoice submitted on {0:D}
exceeds the
          maximum billable amount for Project {1}
({2:C}) and will
          require special approval!"
```

The "D" and "C" values are format strings defined in C# to indicate date and currency values respectively. In the user interface, the lazy evaluation hook was configured to initiate a re-evaluation based on the presence of the double dollar signs. The translated value was looked up, the dynamic data was parsed according to the established convention, and the final string was constructed using the locale conventions of the current user.

In this way, any rule modified to use the above conventions and syntax could show up correctly in different languages and formats depending on the user locale. At the same time, any rule not adhering to the new syntax or conventions would be evaluated according to the original behavior of the rules engine, therefore ensuring perfect backwards compatibility. There was also discussion of adding support for these conventions and the rules resource file into the rules engine development application, but it was deferred due to time constraints and scheduled for a later release.

4 Conclusion

In the end, the project was completed on time, and neither the client nor their customers were required to modify their existing rules. Two principles proved crucial to accomplishing this. First, the insight of treating rules engine output as an intermediate format to be evaluated further at runtime rather than treating it as final output allowed full internationalization of dynamic data with a minimum of code changes. Second, the deferment of localizable data evaluation as long as possible enabled the evaluation to occur in the correct locale context. Applying these basic principles kept the overall code changes to a minimum in terms of scale and risk while also providing backwards compatibility.

Acknowledgements. The author would like to acknowledge the project team for delivering the project that is the focus of this paper, as well as Brenda Hall and Chris Durand for their many invaluable comments and suggestions during the editing process.