AN APPROACH FOR TOTALLY DYNAMIC FORMS PROCESSING IN WEB-BASED APPLICATIONS

Daniel J. Helm, Bruce W. Thompson
The MITRE Corporation, 1820 Dolley Madison Blvd, McLean, VA 22102, USA
Email: dhelm@mitre.org, bthompso@mitre.org

Keywords: web-based computing, database applications, forms processing, dynamic pages

Abstract: This paper presents an approach for dynamically generating and processing user input form variants from metadata stored in a database. Many web-based applications utilize a series of similar looking input forms as a basis for capturing information from users, such as for surveys, cataloging, etc. Typical approaches for form generation utilize static HTML pages or dynamic pages generated programmatically via scripts. Our approach generates totally dynamic forms (including page controls, presentation layout, etc.) using only metadata that has been previously defined in relational tables. This significantly reduces the amount of custom software that typically needs to be developed to generate and process form variants.

1. INTRODUCTION

Many database-oriented applications provide user interfaces that often times are comprised of large numbers of input forms. There are two general approaches for creating user input forms in these kinds of applications:

1. Creating static forms made up of text, images, and HTML formatting tags
2. Developing software that generates dynamic forms using data stored in databases

The static form approach typically requires much manual labor to craft the HTML pages by hand. When large numbers of page variations are required in an application, this approach can become very error prone and a maintenance nightmare. The dynamic form approach, on the other hand, utilizes software such as CGI-scripts [1], Cold Fusion [2], or Active Server Pages [3] to programmatically generate much of the form content using data read from a database. Dynamic forms generated in typical applications are generally comprised of both static and dynamic content. The basic layout and presentation of the form such as the controls (text fields, list boxes, etc.) and their positioning logic are statically defined in the form processing scripts. The dynamic portion of the form consists of the information read from a database that supplies the data element values for the different page controls. Although better than the totally static method, this approach can involve considerable software development to define the many scripts needed to generate and process the form variants.

Our approach, however, utilizes a framework that supports completely dynamic forms (including page controls and layout) created from metadata stored in a series of relational tables. This simplifies the coding required to generate large numbers of forms for certain application types. We applied the concept on a prototype system developed for a customer project that is used to collect large quantities of information on complex information systems as a basis for determining system-to-system interoperability requirements and estimations. Because of the large number of "questionnaires" needed to support very detailed information collection and resolution, we greatly benefited from this totally dynamic form processing technique that reduced the amount of custom software we had to develop.

2. DYNAMIC FORMS CONCEPT

The basic concept is to represent a form via metadata that defines a basic two-dimensional layout. Figure 1 provides an example of our user

interface. The left frame provides a listing of various functions that can be performed, such as requesting a form or generating a report. The right frame is used to display the output of a selected function. The example shows a fragment of a user input form used to collect information about a system's capabilities. A form is comprised of a

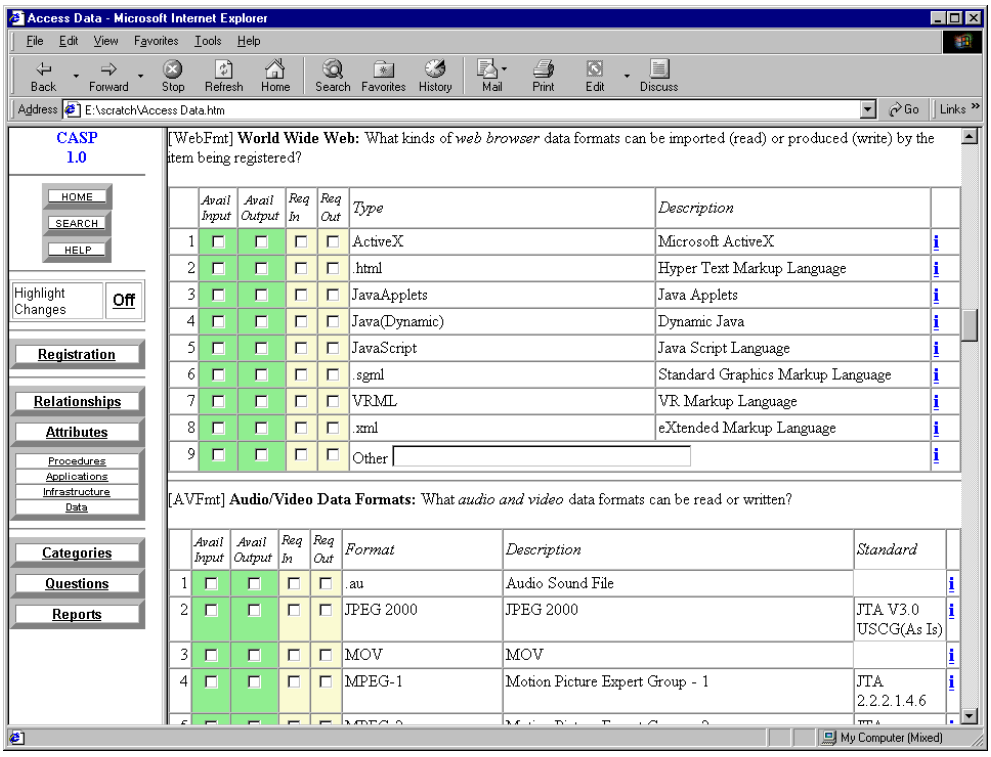

Figure 1. User Interface

series of questions, question options (numbered rows), and option choices (HTML controls). These questions and options can be shared across many forms to support different (overlapping) questionnaires. The user can request a form that is associated with a particular topical area. There are a number of overlapping form categories that can be selected, so it was important to dynamically generate all form content to avoid developing redundant pages and software. Figure 2 shows a data model of the relationships between forms, questions, options, and choices. In addition to defining the relationships between the various form elements, the database schema also supports common HTML control types (checkbox, text field, label, list box, text area, etc.) for choices. This information is stored in the "choice" table. Each major object type (survey/form, question, option, choice) is associated with a particular relational table. Along with the metadata-oriented tables, we also have a table that stores information collected via the forms. This "data store" table has as its primary key, the system identifier and choice identifier associated with a specific element answered by a user. When forms are displayed in edit mode, this table is also consulted to pre-fill the element values. Since choices (via their options) can be shared across different questions and forms, all forms referencing the same option/choice will be "given credit" for the values entered.

3. PROTOTYPE SYSTEM

We developed a web-based prototype system that is used to collect large amounts of information on complex information systems as a basis for evaluating potential system-to-system interoperability capabilities and limitations. The prototype system was developed in the Cold Fusion language and can interface with any ODBC-

compliant database (e.g., MS Access, SQL Server, Oracle, etc). The system is comprised of three major

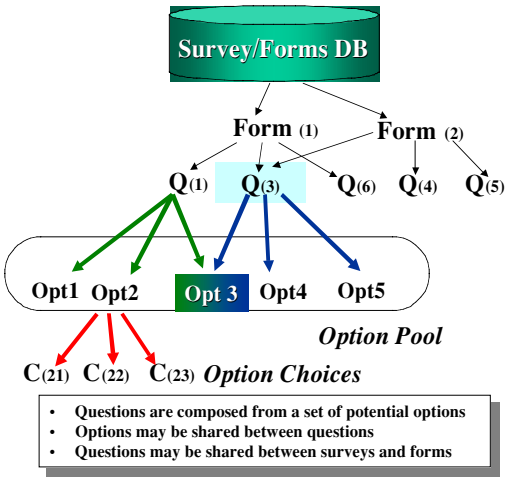

Figure 2. Data Model

modules: administrative, information collection, and report generation. The administrative module is used to populate and maintain metadata in various relational tables. This metadata includes various topics and categories specific to the application as well as the data that defines the survey questionnaires (questions, options, choices). The information collection module (shown in Figure 1) provides the basic end-user interface supporting data collection. The report generation module generates various reports used to evaluate system-to-system interoperability capabilities of the systems for which information has been collected. Although the application was developed in Cold Fusion, the dynamic form concept can be instantiated in other common programming languages such as ASP and Perl (CGI-script). The prototype system has been used for two years and the dynamic form concept has paid off in terms of significantly reducing the software development and cost required to maintain the application, in particular when form requirements and content has changed over time.

Figure 3 shows pseudo code that describes the basic algorithm used to dynamically generate a form. In most cases, places where SQL calls are made are left abstract and are preceded by a "==>" in the code. Program variables are delimited by #'s. When a user requests a particular form, a single Cold Fusion script is invoked to generate the page. The script will first query the "data store" table to pull back the previously entered values for any choices on the form to be displayed. These values are stored into dynamically created variables, whose names are encoded using the corresponding choice identifier. The metadata tables are then consulted to retrieve the question, option, and choice related information needed to build the form.

```
1:   ==> select previously entered values from "data store" table
2:   Loop over returned choice values
3:      // create dynamic variable to hold next value
4:     "variable_#choice_id#" = #choice_value#
5:   End Loop over returned choice values
6:   ==> select questions associated with form category
7:   Loop over returned questions
8:      write question "prompt" text
9:      ==> select column header metadata for next question
10:     Loop over returned column headers
11:        write next column header
12:     End Loop over returned column headers
13:     // get metadata for next question to be rendered
14:     ==> Select *
15:     From question_option_table qot, option_table ot, choice_table ct
16:     Where qot.question_id =  #next_question_id#
17:       AND qot.option_id = ot.option_id
18:       AND ot.option_id = ct.option_id
19:     Order By qot.option_question_order, ct.choice_order
20:
21:     Loop over returned question metadata
22:      If new option Then
23:        start next option (row number)
24:      End If
25:      // Write next control element (choice) for option;
26:      // Set if value previously stored in dynamic variable above;
27:      // Encode control name as "var_#choice_id#_#choice_type#"
28:      Switch #choice_type#
29:        case "radio button":
30:           write radio button control
31:           break
32:        case "checkbox":
33:           write checkbox control
34:           break
35:        case "select":
36:           write select list control
37:           break
38:        case "text":
39:           write text field control
40:           break
41:        case "text area":
42:           write text area control
43:           break
44:        case "label":
45:           write label (static text) control
46:           break
47:      End Switch
48:    End Loop over returned question metadata
49:  End Loop over returned questions
```

Figure 3. Form Generation Pseudo Code

```
1: ==> Delete previously submitted elements from "data store" table
2: Loop over controls submitted on form
3:    If (#next_value# = Evaluate(#next_control#)) <> "" Then
4:       ==> insert next choice control value in "data store" table
5:    End If
6: End Loop over controls submitted on form
```

Figure 4. Form Database Store Pseudo Code

As the form is being dynamically built, the previously entered choice values (stored in dynamic variables) are used (when defined) to pre-set the HTML control values being rendered. After a user fills/modifies the values on a form, a "submit" button can then be clicked to store the answers. A small module in the same script that rendered the form will then be invoked to store the entered values in the "data store" table (see Figure 4). The module first deletes any data that had been previously submitted on the form. This deletion operation simplifies form updates and in particular the processing of checkbox controls which are not transmitted unless they are checked. The module then loops through the passed form elements and will store one record per element in the "data store" table. The HTML element names were encoded with the corresponding choice identifier and choice type, so this simplifies any special processing needed to format specific element values during storage processing.

To be able to delete existing choice values and insert new values into the "data store" table, the module needs to know the form element names that were submitted. These form element names can be determined in a number of ways. One technique is to pass a hidden field that contains a comma delimited list of form element names. This hidden field can be built during the form generation logic. Another option is to pull/parse the names from the standard CGI variable called "QUERY_STRING" that is typically available to an invoked script. Finally, in the case of Cold Fusion, one can access the "fieldnames" variable that contains the list of names submitted from a form.

4. CONCLUSION

This paper briefly discussed a totally dynamic form generation concept that has been successfully used in a complex data collection system. Many database centric applications utilize a series of similar forms as a basis for collecting information from a group of users. Providing a means to reduce the amount of software development time and maintenance required to generate and process such forms can have very high payoff in many enterprise-wide web-based applications.

REFERENCES

Gundavaram, S., 1996. CGI Programming on the World Wide Web. O'Reilly.

Forta, B., 1998. The Cold Fusion Web Application Construction Kit, Second Edition. Que Corporation.

Plourde, W., Slater, B., Slater, W.F., Bass, C., 2001. ASP 3.0: The Complete Reference. Osborne.