A Generic data collection system through WWW forms, based on a Python OOD program.

Benny Shomer

EMBL outstation - the European Bioinformatics Institute
Hinxton Hall, Hinxton, Cambridge CB10 1RQ, **UK**

bshomer@ebi.ac.uk
http://www.ebi.ac.uk/

Tel: +44 1223 494437
Fax: +44 1223 494468

Abstract

One of the problems associated with modern data management is the efficient collection and validation of data from providers of information. WWW browsers, able to support forms provide an excellent, machine independent interface for collecting data through the network. When the information to be provided is rich and complex, a robust program on the server side should provide the user with a friendly interface that breaks the complex task into smaller, more manageable tasks and on the other hand act as a powerful pre-filter, scanning errors on-line.

Albeit some limitations which are a consequence of the server-client philosophy, object oriented programs are especially efficient for interfacing with WWW forms.

Described here is a generic, object oriented data submission system, entirely based on the Python programming language. This system can be easily accommodated to serve several data submission schemes with a relatively short development and implementation time. This system already serves successfully in the EMBL outstation - the European Bioinformatics Institute (EBI). It caters one of the world's largest biology databases - EMBL as well as a few others.

Introduction

The EMBL outstation - EBI (1), maintains one of the world's largest databases of nucleic acid (gene) sequences. The information is provided by research laboratories around the world. Each data record is very complex and contains a large amount of information which accompanies the sequence. Therefore, the process of submitting data to the database is very demanding and complicated.

Some of the common ways of providing information are by editing a text based form , or using a dedicated program which runs on a personal computer and interactively guides the submitter in the steps of submission and as an end result, creates a formatted text file. The text file is then sent by email. There are drawbacks to these methods. Editing a text file is not interactive and there is no machine based process to validate the completeness of data. The standalone program on the other hand, requires re-writing of the code for each platform. It is also very difficult to keep the program up-to-date with the dynamically developing database. These drawbacks led to a search for better ways of interactive data collection.

The recent development of WWW based data entry forms opened a door to an excellent interface between providers and maintainers of information. WWW forms have many advantages. They provide a uniform graphic user interface. They are machine independent and they provide means for a constant, online update and maintenance by the server administrator.

HTML documents, and especially forms lend themselves gracefully to object oriented development. The fact that each element of an HTML document is created by a standard primitive fits well with the concept of objects. This is especially evident in form elements that have a "TYPE" primitive in their tags.

Processing of the data stream which is provided by a submitted WWW form is done by a CGI script. When the data submission process is complicated and involves many validation steps of field contents and many forms, a simple script is not sufficient and a need for a high-level program arises. The Python programming language is especially suitable for writing such programs. Python's unique combined ability to serve as a fast scripting language and at the same time as a high level object oriented development language turns it into a very powerful development tool for WWW development.

This article describes a data submission system that has been developed by the author at the EBI. The system is based on a collection of Python objects which define it's basic properties and the basic steps in the cycle of acquiring, validating and submitting information. The objects are defined in a generic way which enables the same basic system to cater different data collection schemes. The design of the system makes it easy to support various concepts such as data hiding, dynamic creation of forms, preservation of data items for later use and providing context sensitive help. A data submission system for the EMBL database has been released to the public in May 1995 and it has been operating successfully since then, providing more than 850 data entries so far.

Methodology

Service classes.

web

web is defined as a basic class which provides an easy interface to generating HTML primitives. The various HTML primitives are defined as methods of this class. Some of the primitives are very simple, e.g. for invoking headers, anchors, figures and ordered or unordered lists. Other methods perform somewhat more complex invocation of form elements, e.g. predefined buttons, radio buttons, checkboxes, data input fields and option lists (See figure 1).

valids

valids is a general service object holding methods that validate a given piece of information and returns a validity code (i.e. 0 for non-valid and >= 1 for valid) (See figure 2). This class can be used directly by invocation or as part of a multiple inheritance invocation. The methods provide checking of information in various levels of complexity. Some of the methods provide simple checks such as ensuring that the information is safe and there is no attempt to invoke a shell with the text entered, or ensuring that an entry is only text or only numeric. Other methods perform more specific tasks (e.g. ensuring an entry is a valid nucleic acid sequence entry) or more complex (e.g. checking the syntax of an email address and assuring that the domain provided is a valid one)

GFOD

GFOD stands for Generic Forms Object Definition. It is a collection of the basic classes with their data fields and methods that together form the basis for a data submission process and to inheriting instances.

item

This is the basic unit for implementation of any information ***item*** to be obtained from the submitter. The basic elements of any ***item*** class are the 'key' and the 'value'. Various data fields provide the ***item*** with characteristics that control it's visual implementation in the form. An ***item*** class has the basic methods which control it's validation and ability to report itself as valid, the ability to store it's elements in a file (given the open file handle). The ***item*** has an empty errors list. In case of a failed validation, the error list is filled with the error messages and is presented together with the ***item***. The ***item*** has three different presentation methods. When the ***item*** is presented for the first time, or is non-valid, it is being presented in the form as defined in the 'present' method. When the ***item*** is valid it uses the method *'present_hidden'* to present itself in the form as a hidden element. This is one concept of data hiding which is practised here, i.e. a valid data ***item*** should not be presented any further in the form. The third method controls the way an ***item*** present's itself in a special 'final validation' form (to be discussed later). The actual implementations of the system use classes that inherit '***item***' and suit it for more specific needs. Examples for such elements (See figure 3) may be a *hidden_item* (always valid, so always hidden), an email item, a radio cluster item that provides a cluster of radio buttons for the same data field, an option list item, items for mandatory text or numeric items etc.

form_obj

Each form in the sequence of submissions (Besides the form which initiates the submission process and two specialised forms that conclude the submission process) is an instance of a '***form_obj***' class. The ***form_obj*** owns and controls a list of '***item***' objects. Each ***form_obj*** with it's ***item***s constitutes an integral entity in the submission sequence. The form object has a special data dictionary that maps the ***item***s. If the ***form_obj*** instance is initiated with a data dictionary, it's ***item***s are initiated as having the appropriate values (e.g. when restoring the form contents from a file), otherwise, the form ***item***s are initiated with their default values. Each form is identified by a 'level' data field which is actually it's linear order in the sequence of forms of the whole submission process. The ***form_obj*** class is equipped with methods that enable it to store and retrieve it's information to/from a temporary file. This is important especially due to the fact that personal systems and even more, WWW browsers are highly prone to crashing. The ***form_obj*** validation process is a more complex one, controlled by three methods. The 'validate' method calls each of the ***item***s to report their valid status. If any of the ***item***s is non-valid, the ***form_obj*** is set to non-valid. Control is then handed over to a virtual method *more_validate*. This empty method provides inheriting classes with the opportunity to practice an extra step of validation according to logical relations between various ***item***s. To conclude, if the ***form_obj*** status is valid, the *finally_validate* method returns control to the calling object (the ***session***), but if the status is non-valid, the method takes control and invokes the presentation of the form. The form has two methods of presenting itself. The major one *'present_form'* invokes the presentation of a WWW form and then calls each '***item***' instance's presentation method to implant itself according to it's validity status (i.e. if the ***item*** is valid it is presented as hidden). The other presentation method enables the ***form_obj*** to present it's data in the special step of final validation before concluding the submission.

session

The whole submission process is owned and managed by the class '***session***'. This class owns a list of '***form_obj***' instances to scroll through the submission process, calling each form in it's turn. The process, which can be invoked by many users at the same time is controlled with the aid of two identifiers. At the beginning of the submission process, the submission is being assigned with a unique identifier which is implanted into each of the forms. Each of the form objects is identified in the sequence of submission by a 'level' which is actually it's position in the objects list. There are also special levels according to the stage of submission. The process which takes place for all the ***form_obj*** (or it's inheriting) instances is a simulation of an event loop. Initially, the form is set to non-valid and is displayed.

Following submission, the '***session***' object identifies the current level and accordingly calls the form's validation method. If the form is valid the ***session*** invokes the display of the next form in it's objects list. If the form is still non-valid, the form itself takes control and displays itself. This contradicts the common concept of an event handling hierarchy, but is evidently a consequence of the fact that server processes are killed every time a document is sent back to the client.
The ***session*** is equipped with an empty method called *'expand_dynamic'*. This method is called (even as an empty one) in the appropriate places, to enable later implementations of a dynamic expansion of the ***form_obj*** list. Thus, a user can select adding more instances of a certain ***form_obj*** class according to the need. An inheriting class that needs a dynamic expansion of forms must encapsulate the expansion method in the *'expand_dynamic'* method, so it can always be called when needed. When the last form in the '***session***' object list is reported back to be valid, the ***session*** invokes a special form for final validation of the information. This is done by calling the *'present_contents'* method of each form. The resulting form has a checkbox associated with each '***form_obj***' instance. This form produces a corrections list. When the form is submitted, it is identified with a special 'level'. If it provides a non-empty corrections list, the ***session*** iterates again through the selected forms by setting them as non-valid thus forcing them to present themselves for corrections. When the corrections list is empty the ***session*** *'produce_output'* method is called to produce the output of the submitted information. Following production of output, the user is allowed to select forms that contain data to be used again in a consecutive submission cycle. This is done by the ***session*** class, which has a *'repeat_forms'* list of allowed levels to be repeated. After the user has selected the required form levels, the ***session*** dumps a virtual memory file which contains the appropriate *key=value* combinations for preservation of information. Now, on invocation of each '***form_obj***' instance, the ***session*** will compose a data dictionary which is handed to the form's *'init'* method. When such a dictionary is present, the form will invoke the '***item***' instances with the appropriate values as default.

Discussion.

In a modern world where information is produced and accumulated in very high rates, it is important to device data collection mechanisms that will be friendly and efficient, device independent, yet capable of filtering out as many errors as possible and sort information according to specifications before a human interpretation step is required. WWW clients that support forms are a very powerful interface for collecting information from users through the network. The advantages of the forms system are many. The WWW browser provides an easy way of enjoying a graphic user interface, while being absolutely machine independent, since WWW browsers exist for all the common platforms.

This article described an information submission system that operates based on an httpd server and is written entirely in the Python programming language. Python was found to be extremely suitable for the development of WWW service programs. Python's unique combination of capabilities to perform as an interactive interpreter, as a powerful scripting language and a high level object oriented design tool provides an excellent development tool for WWW associated programs.

The system was developed in the spirit of an object oriented program which follows the event handling loop scheme. This was easily obtained by using Python's ability to store pointers to objects in a dynamically linked list. Thus, each form object holds a list of item objects which it uses to iterate while calling each object's methods. In the same way, the session object holds a list of form objects, iterated in the same fashion. Dictionaries that map data and that can be passed as arguments for a method provide the ability to initialise the form objects with default values that were provided by the user.

A major limitation exists when developing for WWW environments. The fact that the server's process is killed once an information page is dismissed to the client breaks the continuous flow of the program. Therefore, solutions must be devised to resume the process after breaking. In the EBI information submission system, each form object dumps it's data dictionary as a temporary file. This temporary file is to be used later to restore the form object's data dictionary, resuming it's status to what it was when the process was terminated.

During the time of developing and running the system, several limitations of WWW forms were characterised. Some of these limitations create severe problems when WWW browsers are being used for performing more complicated and demanding tasks of interfacing and exchange of information. One such typical problem occurred while constructing an options list for selecting of information regarding the species to which the information relates, since the possible list is very long and there is a limitation of displaying a maximum of 28-30 options in a list. This is very limiting since only the most popular species can be included in the options list, while the rest must be typed into an input field. This may lead to many errors. Species can be grouped according to families, orders etc. This enables creating a nested options list, that reduces the length of each list and even more, introduces a logical structure into the array, reducing the risks of typing or other errors. Currently, the HTML standard does not support the concept of nested options lists. Another drawback is that transferring such a large nested list through the network requires much net traffic, which reduces the system's efficiency. One possible solution to this combined problem would be to write a special applet for the Grail browser. The applet code will be capable of transferring a compressed list, possibly saving it on the local disk for further uses (like the browser's caching for graphics) and to present the nested list, implanting the selected value into the message.   The data validation process is crucial. When the data submission process is complicated and involves many steps, the submission of a form which contains an error may be a frustrating step, especially when network load is high. If the browser could check the entry for some logical delimiters (e.g. not-empty, numeric contents and range etc.) much of the error filtering would be done even before the information leaves the submitter's browser. This too, can be done by implanting applets in a Grail client.

Another common problem occurs when submitters have fairly large sequences to submit. This creates a technical problem of cutting and pasting the sequence, which is prone to errors. Also, there is a technical limit for most browsers to transfer more than 23Kb of information from a single window. A very efficient solution to both of these problems would be the ability to attach a file with the submission. This is not implanted yet in the browsers and it is probably another application that can be implanted in the Grail system.

The future developments will obviously depend on the success in exporting binaries of Grail for all the commonly available platforms and especially personal computers. There is no doubt that the popularity the WWW browsers have gained was primarily due to their availability for PC and Macintosh platforms.

Python's usefulness as a robust development tool for Internet applications is doubtless. It's flexibility in providing interactive programming and scripting environment as well as a high level interface to object oriented design will probably place it in the near future in a distinguished position among the most popular Internet development tools.

References

1. **David B. Emmert, Peter J. Stoehr, Guenter Stoesser** and **Graham N. Cameron**. (1994) The European Bioinformatics Institute (EBI). *Nucleic Acids Research, 22:3445-3449*

Figures

```
1.
>>> from WWW import web
>>> w = web()

2.
>>> w.initialize('This is a test document')
Content-type: text/html

<HEADER>
<TITLE>This is a test document</TITLE>
</HEADER>
 <BODY>

3.
>>> print w.anchor('http://www.ebi.ac.uk/','The EBI home page')
<A HREF="http://www.ebi.ac.uk/">The EBI home page</A>

4.
>>> print w.fig('../icons/spam.gif')
<IMG SRC="../icons/spam.gif" ALT="IMAGE" ALIGN="MIDDLE">

5.
>>> w.form('http://www.ebi.ac.uk/htbin/embl_subs.py')
<FORM METHOD="POST" ACTION="http://www.ebi.ac.uk/htbin/embl_subs.py">

6.
>>> w.button_submit()
<INPUT TYPE="submit" VALUE="Submit">

7.
>>> print w.lineinp('ADDRESS',40,'My default address.')
<INPUT TYPE= "TEXT" NAME= "ADDRESS" SIZE= "40" MAXLENGTH= "20"
VALUE= "My default address.">

8.
>>> mylist=['hats','socks','shirts','pants','jackets','ties']
>>> print w.option('clothing',mylist,3)
<SELECT NAME = "clothing">
<OPTION> hats
<OPTION> socks
<OPTION> shirts
<OPTION SELECTED > pants
<OPTION> jackets
<OPTION> ties
</SELECT>
```

Figure 1.
The class "web" - an example of the use of some of it's methods. **1**. The class instance is invoked. **2**. Initialisation of an HTML document. **3.** Anchoring of a URL. **4**. Providing an image. The ALT and ALIGN fields have defaults that can be altered. **5**. Initialisation of a form. The method defaults to POST and may be altered to GET by the invocation. **6**. Instant invocation of a submit button. **7**. Creating a line input of a form with a default text. **8**. Creating an options selection list with a pre-defined selection.

```
1.
>>> from validations import valids
>>> v = valids()

2.
>>> v.safe('!#rm *')
'#rm *'

3.
>>> v.is_text('This is a text')
1

4.
>>> v.is_numeric('number 69')
0

5.
>>> v.is_email('bshomer@ebi.ac.zz')
2

6.
>>> v.is_na_sequence('actaacggatacnnacagt')
2
```

Figure 2.
The class "valids" - an example of the use of some of it's methods. **1**. The class instance is invoked. **2**. Ensuring that there is no attempt to invoke a shell. **3.** Checking an item contains only ASCII128 characters. **4**. Checking an item is numeric. **5.** Checking an email address. The return value implies that the syntax is correct but the country domain does not exist. **6**. Checking a nucleic acid (DNA) sequence. The return value implies the sequence contains an ambiguous character which is permitted in sequences only in special cases.

```
1.
email = subs_obj.email('email', '', 'email_help.html', 40, 80)
email.item_second_title = 'e-mail address'

2.
journal = Goption('journal', '','journal_help.html',3)
journal.item_title = 'Journal:'
journal.line_break = 0
journal.option_list(['Biochem. Biophys. Res. Commun.', 'Biochem. J.',\
'Biochemistry', 'Biochim. Biophys. Acta', 'Cell', 'Curr. Genet.'])

3.
viral_circular = __Gcheckbox('viral_circular')
viral_circular.item_second_title = 'circular'
viral_circular.line_break = 0

4.
dtemp = Goblignum10('dtemp', '', '', 85, 100)
dtemp.item_title = w.hr(1) + 'Denaturation:'
dtemp.item_second_title = ' dC. '
dtemp.item_line_title = 'Temperature: '
dtemp.line_break = 0
```

Figure 3.
Examples for instance invocations of classes that inherit 'item'. **1**. An email item. It has a unique validation method that checks the email syntax and the existence of the domain. This item's instance uses a 'item_second_title' data field which causes it's title to be displayed following the input field. The parameters that are being passed to the __init__ method are the item's key, value, name of a help file (if it is given, an anchored help icon is displayed) and the lengths of the field and maximal length of input. **2**. An option list item. The item has an associated 'option_list' to be displayed. The parameter '3' given following the help file determines the option to be selected by default. The 'line_break' data field determines if the item can be displayed on the same line with other items or not. **3.** A special case of an item that supports a concept of data hiding. This item is initially hidden. It becomes non-valid (and thus visual) only if the submitter provided information that is associated with the item's field. For example here, only if the submitter submits the sequence of a virus, this checkbox will be presented, requesting further details about the nature of the virus. **4.** An obligatory numeric item with minimum and maximum values. The item is non-valid as long as it's value is empty or out of the range given. This overlap in validations may seem as redundant, but since the minimum/maximum fields may be omitted (any range is allowed), this is not the case.