UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique
UMR CNRS 7523 Heudiasyc

OMAS WEB EDITOR
Programming

Jean-Paul Barthès

BP 349 COMPIÈGNE
Tel +33 3 44 23 44 23
Fax +33 3 44 23 44 77

Email: barthes@utc.fr

N297 v1.0
February 2014

Warning

The document contains information to prepare an application to be able to use the OMAS Web Editor.

Keywords

OMAS, Web Editor

Revisions

Version	**Date**	**Author**	**Remarcks**
1.0	Feb	Barthès	Draft

1 Introduction

The OMAS editor allows to directly modify objects that are kept in agent space. It works from a web browser provided the application has been prepared to allow its use. The present document explains what must be done to allow the OMAS editor to work with a given application.

2 Requirements for Editing Objects

An object in an agent space is usually an instance (individual) of a class (concept). We say that the agent is the owner of the object. The object may be kept in core or saved in a database. This does not change the way it is edited. An object will be edited in a page of a browser. Such a page is a form, allowing to modify the object being displayed, as for example in Fig.1.

Figure 1: The N-4 news item to be edited (the red fields are read-only)

We thus need to tell the browser how to build such pages and how to fill them. In a standard approach the editing actions are generic (i.e. the same for all objects):

- **Abort editing this object**: the edit page is redisplayed with the original values of the object, i.e. those kept in the "initial object" field of the user entry. A message is posted.
- **Create/Update**: the web-edit-commit function is called. The new values are compared with the values stored in the "initial object" field of the user entry. If they are identical, the edit page is redisplayed with a message saying that nothing was changed. If they are different, then an update program is built for adding, removing or modifying the different properties of the object and shipped to the owner agent to execute. The agent returns the local ID of the object (which is new for a creation) and a list of messages giving some information about the creation of ancillary objects.
- **Edit new object**: the selection form is called. We keep the same owner agent, class reference but do not provide data. Of course they can be overwritten.
- **Again**: the edit page is redisplayed ready for creating a new object of the same class

- **Delete**: the owner agent is called to delete the posted object. Note that no confirmation is required since we select the Delete radio button AND have to click the Execute button.

Updating the content of an object can be done in a generic fashion. However, creating a new instance is not so easy and needs some help from the application.

The following sections detail what must be done, namely:

- how to specify a page format
- how to add needed methods in the ontology file
- how to write create skills

We will take examples from the NEWS application.

3 Specifying a Page Format

Specifying a page format is done by putting a defeditlayout macro in the agent file that owns the class of interest.

3.1 Examples of Pages

A simple example of a page description containing information about a person is the following.

```
(defeditlayout "Person" :STEVENS-NEWS
  ((:row :header "Name(s)*" :name "name")
   (:row :header "First name(s)" :name "first name")
   (:row :header "Initials (e.g. JD for John Doe)" :name "initials")
   (:row :header "Published (read only)" :name "published" :read-only t)
   )
  )
```

A more complex example is the page shown Fig.1 defined as follows:

```
(defeditlayout "News item" :STEVENS-NEWS
  ((:row :header "ID (given by the system, read-only)" :name "id number"
         :read-only t :initfcn create-id)
   (:row :header "Title*" :name "title")
   (:area :header "Author, e.g. Barthès:Jean-Paul, Ramos ..."
          :name "author" :rows 1 :cols "60%" :read-only t
          :path ("author" "person") :initfcn create-author)
   (:row :header "Category (max 1)" :name "category" :path ("category" "Category" "label")
         :if-does-not-exist :ignore :selfcn sel-category :selprop "label" :type :mln)
   (:row :header "Indexes" :name "indexes" :path ("indexes" "Index")
         :if-does-not-exist :create)
   (:area :header "Text" :name "text" :rows 3 :cols "60%")
   (:row :header "Creation date (read-only)" :name "creation date" :read-only t
         :initfcn create-date)
   (:row :header "Modification date (read-only)" :name "modification date" :read-only t
         :postfcn create-date)
   (:row :header "Published (read only)" :name "published" :read-only t)
   (:row :header "Archived (read-only)" :name "archived" :read-only t)
   (:user-action :header "Publish..." :name "user-action" :value "publish")
   )
  )
```

We can see from the code that each field of the web page is defined either as a row or an area with some additional options. Two options are compulsory: (i) :header that defines the label of the field; and (ii) :name that defines the internal name of the parameter that will be associated with the content of the field. Among the other possibilities are :initfcn, :path, :read-only, and more.

3.2 The defeditlayout Options

The meanings of all the options of the defeditlayout macro are summarized in the following table.

Option	Meaning
:row	gives an approximate width of the field wrt the width of the web page
:area	specifies that the field may have more than a line, which is indicated by the :rows option
:user-action	defines an action specific to the application. :name in that case should be "user-action"
:cols	specifies the hight in lines of an area
:header	text labeling the field
:if-does-not-exist	used when the field corresponds to a relation and the content points to an object that does not exist. Tell the system what to do in that case, either :create or :ignore
:initfcn	name of a function that will be called to execute some precondition related to the corresponding field, e.g. to fill with a default value
:name	name of the internal parameter associated with the content of the field. It is a good idea to give the name of the property associated with the field whenever possible.
:path	when the field corresponds to a relation, gives the path corresponding to the value that will be posted in the field
:postfcn	indicates a function that will be executed on the value of the field as a post-condition
:read-only	specifies that the field is read-only
:rows	specifies that the field is a simple one line field
:selfcn	specifies a function that will produce a list to choose from a pull down menu
:selprop	used in conjunction with :selfcn to point to the property that will produce the values of the pull down menu
:type	used to describe the type of value to be posted in the field. Mainly used to indicate a multilingual name (:mln)

One must note that each field in a page corresponds to an attribute or to a relation of the displayed object. In the latter case the value that will appear in the field is either a summary of the neighboring object (successor in MOSS terms) or a value of one of the attributes of the successor, like the name of an organization or the acronym of this organization. The two cases are distinguish by the :path option. In the first case, the path does not include any final attribute, e.g.

```
("author" "Person")
```

In the other case

```
("category" "Category" "label")
```

3.3 Examples of Fields

This section contains examples of how to specify a field for displaying an object. This depends mostly of the type of property on which the field depends. Attributes lead to simple specifications, relations are more complex. This section comments examples of both cases.

3.3.1 Displaying Attributes

Ex.1 Displaying attributes is relatively simple. For simple values to be posted as it it is enough to use a minimal format, e.g.

```
(:row :header "Title*" :name "title")
```

Ex.2 If we want the value to be read-only, then we write

```
(:row :header "Published (read only)" :name "published" :read-only t)
```

Ex.3 If the value is a read-only value with an initial computation then we write

```
(:row :header "Creation date (read-only)" :name "creation date" :read-only t
      :initfcn create-date)
```

Of course we must provide the create-date function, here a simple deed

```
(defun create-date ()
  "creates a date as jj/mm/yyyy"
  (list (moss::get-current-date :time (get-universal-time))))
```

3.3.2 Displaying Relations

Relations are more complex to handle, since we must print something about the object being linked to the object we are editing.

Ex.1 In the simple case of writing the name of the person who is the author of the object we are editing, we write:

```
(:row :header "Author, e.g. Dupond" :name "author" :path ("author" "person" "name"))
```

The :path option describes the link between the object we are editing and the successor along the author link. OMAS will use the option to retrieve the value of the person name.

Ex.2 If we want to print a summary of the linked object, then we just omit the last attribute, e.g.

```
(:row :header "Author, e.g. Dupond" :name "author" :path ("author" "person"))
```

In that case, because no attribute is specified, OMAS will execute the =summary method, instance method of person to compute the right value to print, e.g. "Barthès: Jean-Paul" or "Fujita: S." However, this brings up a problem when we want to modify this field by adding new values. Say we want to add "Dupond: Jean", then OMAS will have to look up in the data base to find out if Jean Dupond exists. But OMAS needs to know what is the name and what is the first name or sometimes the initial(s) of the first name. To allow that one has to define a method =parse-summary that is the inverse of the =summary method. This will be discussed later.

Ex.3 Like in the attribute example, we can add an initial function to fill in one or more values by specifying a function

```
(:row :header "Author, e.g. Dupond" :name "author" :path ("author" "person")
     :initfcn create-author)
```

Of course we must provide this function that will point to specific persons.

Ex.4 Sometimes the attribute to be printed is a multilingual name, in which case we must tell it to OMAS. This is the case for country or city names that can have different spelling in different languages, e.g. London in English and Londres in French.

```
(:row :header "City name" :name "city" :path ("city" "City" "name") :type :mln)
```

This last option may be combined with the others.

4 Needed Additional Methods

If most of the actions in the editor are generic and do not require to write any special code, some involve specific elements of the application. Some methods are sometimes necessary for processing the content of a field, as was mentioned in the previous section.

4.1 =parse-summary Methods

When a field corresponds to a relation and displays information about an object linked to the object being edited (a successor in the MOSS jargon) without referring to a particular attribute, then OMAS uses the =summary method to compute the value being posted. For example, if we want to display information about persons as

```
Dupond: Jean, Doe:J.
```

We use the following =summary method

```
(definstmethod =summary person ()
  (list (format nil "~{~A ~}: ~{~A~^, ~}" (g==> "name") (g==> "first-name"))))

#|
(send '$E-PERSON.3 '=summary)
("Barthès: Jean-Paul")
|#
```

Now if we want to change this value to

```
Dupond: Jean, Dupont: René
```

Removing Mr Doe and adding Mr Dupont, we must give OMS the possibility to parse the corresponding data. This can be done by declaring a special =parse-summary function, for example in that case

```
;;;-------------------------------------------------------- (PERSON) =parse-summary
;;; initials could be characterized by the fact that their length is twice the
;;; number of dots in the string after removing hyphens "-"  and equal signs "="
;;; But since this depends on cultures we require that first names do not contain
;;; dots

(defownmethod =parse-summary $E-PERSON (val)
  "takes a value like \"Barthès: Jean-Paul\" or \"Moulin: C.\" and returns the ~
   list ((\"name\" \"Barthès\") (\"first-name\" \"Jean-Paul\")) or
        ((\"name\" \"Moulin\")(\"initials\" \"C.\"))
We assume here that the first name does not contain dots like in \"Jean-Paul A.\" ~
as can be found if the middle name is attached to the first name."
```

```
  (let (name first-name initials pos rest)
    ;; val must be a string
    (when (stringp val)
      (setq val (string-trim '(#\space) val))
      ;; extract name first
      ;; get the position of column
      (setq pos (position #\: val))
      (cond
       (pos
        (setq name (string-trim '(#\space) (subseq val 0 pos)))
        ;; extract the rest
        (setq rest (string-trim '(#\space) (subseq val  (1+ pos))))
        ;; now try to figure out if a first name or initials
        ;; if no dot present, first name
        (setq pos (position #\. rest))
        (if pos
            ;; might be initials
            (progn
              (setq initials rest)
              )
          ;; must be first name
          (setq first-name (if (equal+ rest "") nil rest)))

        ;; return description
        `(("name" ,name)
          ,@(if first-name `(("first-name" ,first-name)))
          ,@(if initials `(("initials" ,initials)))))
       ;; if no ":" and not empty string assume it is the name
       ((not (string-equal val ""))
        `(("name" ,val)))
       ))))
```

4.2 Multilingual Names

When a value is a multilingual value, we can define a =parse-summary method that transforms the simple name into a multilingual (mln) format, e.g.

```
;;;---------------------------------------------------------- (CITY) =parse-summary

(defownmethod =parse-summary $E-CITY (val)
  "takes a value like \"New York\" and returns the list
   ((\"name\" (:FR \"New York\")) if :FR is the current language."
  `(("name" (,*language* ,val))))

#|
(send '$e-city '=parse-summary "New York")
(("name" (:FR "New York")))
|#
```

4.3 Abbreviations

If we want to let the user type for example a single letter in a field rather than the full value, we can define a =xi method that will parse and validate the input data, e.g.

```
;;;-------------------------------------------------------- (TYPE DE CONTACT) =xi

(defownmethod =xi ("type de contact" "moss property name" "moss attribute"
                     :class-ref "contact") (data)
(cond ((and (stringp data)(string-equal data "S")) "suivi")
           ((and (stringp data) (string-equal data "R")) "rencontre")
           ((and (stringp data) data))))
```

5 Required Additional Creation Skills

When creating new individuals some operations require a special handling in some application, which means that we cannot use a generic approach. Thus, some specific skills are necessary for creating individuals, in particular when this involves side-effects.

5.1 The :CREATE-OBJECT Skill

Whenever a new instance (individual) of a class (concept) is allowed to be created and has a specific edit page, then we must define a :CREATE-OBJECT method at the agent level. The method is intended to dispatch the creation to more specific methods attached to different classes. For example

```
;;;============================================================================
;;;                            CREATE-OBJECT
;;;============================================================================ skill
;;; called by the default :create-object-skill to create (and possibly link) a
;;; new object. Used to dispatch the creation process to more specific :skills if
;;; they exist

(defskill :create-object :CONTACT
  :static-fcn static-create-object
  :dynamic-fcn dynamic-create-object)

(defun static-create-object (agent message args)
  "function to dispatch the creation of new objects.
Arguments:
   agent: CONTACT
   message: unused
   args: a list of pairs corresponding to the fields of the edit page, e.g.
        ((:data <class-ref> <parms>)
Return:
   the id-ref of the object, e.g. \"$E-INDEX.18\" ."
  (declare (ignore message))
  (let* ((data (cdr (assoc :data args)))
         (class-ref (moss::%string-norm (car data)))
         )
```

```
    ;; check if we are simply creating or if we create and link
    (format t "~2%;========== CONTACT: CREATE-OBJECT")
    (format t "~%~A: static-create-object /args:~% ~S" (omas::name agent) args)

    ;; dispatch creation to the specialized skills. Any class with an edit page
    ;; must have a specific creation skill
    (cond
     ((equal+ class-ref "contact")
      (send-subtask agent :to (omas::key agent) :action :create-contact
                    :args (list args))
      (return-from static-create-object (static-exit agent :done)))
     ((equal+ class-ref "visite")
      (send-subtask agent :to (omas::key agent) :action :create-visit
                    :args (list args))
      (return-from static-create-object (static-exit agent :done)))
     )

    ;; if none of those, error (will be processed by get-answer)
    (static-exit agent :failure
                 :reason (moss::string+ "No creation skill for " class-ref))))

(defun dynamic-create-object (agent message answer)
  "receives an answer (id-ref <message-list>) of the new object, and returns it."
  (declare (ignore message))
  (format t "~2%;<<<<<<<<<< ~A: dynamic-create-object /answer: ~S"
    (omas::name agent) answer)
  (if (car answer)
      (dynamic-exit agent answer)
    (dynamic-exit agent :failure :reason (cadr answer))))
```

Note that the method is a skill of the agent CONTACT. When the agent receives a message to create an object it sends itself a message according to the class of the object, here :create-contact or :create-visit. Since the agent is sending a message and must wait for the answer a dynamic part of the skill is provided. If the answer is NIL then a failure is returned to the caller.

5.2 Specific Creation Skills

Because specific skills are used by :CREATE-OBJECT, they must be given to the agent. For example

```
;;;============================================================================
;;;                           CREATE-VISIT
;;;============================================================== skill
;;; adds a visit by creating it.
;;; We assume that we receive a set of pairs coming from the fields of the
;;; web page, e.g.
;;;   (("pays" . "Zimbawe")
;;;    ("visiteur" . "Mbozza: Albert")
;;;    ("organisme" . "HEUDIASYC")
;;;    ("date de début" . "2/2014")
;;;    ("durée" . "5j")
```

```
;;;    ("commentaire" . "RAS"))

(defskill :create-visit :CONTACT
  :static-fcn static-create-visit
  )

(defun static-create-visit (agent message args)
  "creates a new contact if it does not already exists.
Arguments:
   agent: CONTACT
   message: incoming message (ignored)
   args:  ((:data <class-ref> <parms>))
Return:
   a list (\"$E-VISIT.7\" <message-list>)"
  (declare (ignore message))
  (format t "~2%;========== CONTACT: Entering :CREATE-VISIT")
  (format t "~%;-- static-create-contact /args:~% ~S" args)
  (let* ((parms (cddr (assoc :data args)))
         id message-list)
    ;; here we do not check if contact was already created, we create a new one

    ;; call function to create object, using OMAS generic function
    (multiple-value-setq (id message-list)
      (omas::create-object agent "Visite" parms))

    ;; return the id-ref of the newly created object to the caller
    (static-exit agent
                 `(,(symbol-name id)
                     (,@(reverse message-list)
                         "New Visit was created")))))

#| Test:
(setq parms '(("pays" . "Zimbawe")
              ("visiteur" . "Mbozza: Albert")
              ("organisme" . "HEUDIASYC")
              ("date de début" . "2/2014")
              ("durée" . "5j")
              ("commentaire" . "RAS")))

(static-create-visit sa_contact nil `((:data . ,parms)))
("$E-VISITE.1"
 ("I created a new object: \"Mbozza : Albert\"" "I created a new object: \"HEUDIASYC\""
   "New Contact was created"))

(send '$e-visite.1 '=print-self)

----- $E-VISITE.1
 date de début: 2/2014
```

```
 durée: 5j
 commentaire: RAS
 visiteur: Mbozza : Albert
 organisme: HEUDIASYC
-----
:DONE
|#
```

Note that the skill return a list of two elements: the first one contains the internal key of the object that has been created as a string (this is needed for displaying the object in the web page), and a list of messages explaining what happened during the operation. The input to the skill is the list of fields transmitted by the web browser.

Note also that in order to simplify the code, we use an internal generic function omas::create-object that does the creation. This function can be used whenever there are no special side-effects linked to some of the properties of the object being created.

6 Advanced Options

Sometimes the application requires to do some special actions on the objects being created or modified. For example, in the NEWS application, we want to be able to send a news item to the agent PUBLISHER, to make is available to the outside world. This can be done by using the :user-action option.

```
(:user-action :header "Publish..." :name "user-action" :value "publish")
```

The option create an additional radio button in a row underneath the row of the standard edit actions (Fig.2).

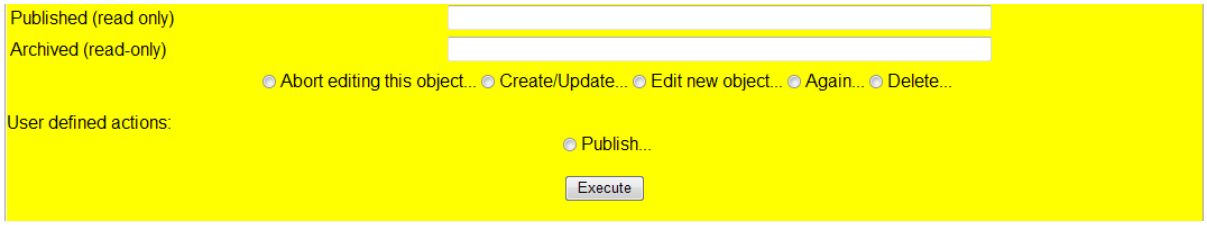

Figure 2: Example of user-defined action button (Publish) on the edit page

OMAS then require the user to write a specific skill corresponding to the user actions, here

```
;;;============================================================================
;;;                          WEB-EDIT-USER-ACTION
;;;============================================================================ skill
;;; called by the server for sending an object to PUBLISHER

(defskill :web-edit-user-action :STEVENS-NEWS
  :static-fcn static-web-edit-user-action
  :dynamic-fcn dynamic-web-edit-user-action)

(defun static-web-edit-user-action (agent message args)
  "function to send an object to PUBLISHER
```

```
Arguments:
   agent: STEVENS-NEWS
   message: unused
   args: a list of pairs corresponding to the fields of the edit page, e.g.
        ((:data <class-ref> <parms> <id>)
Return:
   a list (nil|:ERROR <message-list>)."
  (declare (ignore message))
  (let* ((data (cdr (assoc :data args)))
         (class-ref (moss::%string-norm (car data)))
         (parms (cadr data))
         (ref (caddr data))
         obj-l)

    (format t "~2%;========== STEVENS-NEWS: WEB-EDIT-USER-ACTION")
    ;(format t "~%~A: static-web-edit-user-action /args:~% ~S" (omas::key agent) args)

    ;; remove the "user-action" field from parms
    (setq parms (remove "user-action" parms :test #'equal+ :key #'car))

    ;; produce an object list
    (setq obj-l (omas::parm2objdesc agent class-ref parms))

    ;; send to PUBLISHER
    (send-subtask agent :to (omas::key agent)
                  :action :send-to-publisher
                  :args `(((:data ,ref ,class-ref ,obj-l)(:language . ,*language*)))
                  :timeout 10)
    ;; mark object as saved
    (send (intern ref) '=add-attribute-values "published" '(t))

    ;;get out
    (static-exit agent :done)))

(defun dynamic-web-edit-user-action (agent message answer)
  "we get here the answer from PUBLISHER or from the timeout handler."
  (declare (ignore message))
  ;(print `(SN/ dynamic-web-edit-user-action - received answer ,answer))
  ;; return the answer from PUBLISHER (<message-list>)
  (dynamic-exit agent answer))
```

Note the dynamic function that waits for an answer from the PUBLISHER agent.

7 Calling the Editor from a Web Page

When calling the editor from another web page, i.e. from an application (Fig.3), we assume that we do not need to login. The web page can be called by giving the proper arguments to the HTTP message, i.e.

```
http://<OMAS web server>/editentry?create=t&agent-ref=STEVENS-NEWS&class-ref=news\%20item
```

`&caller=index`

editentry is the OMAS Editor entry form for external accesses, caller specifies the calling page.

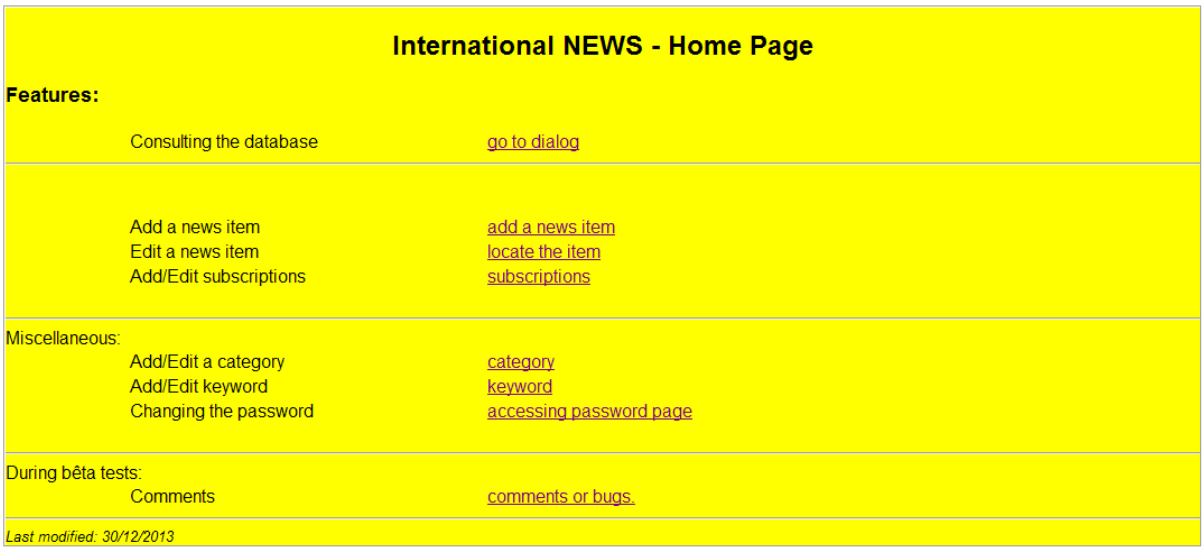

Figure 3: Example of index form from the International News application

If one wants to give more information to locate a particular object (e.g. clicking the locate-the-item link in Fig.4), then ons should label the connections as follows:

```
http://<OMAS web server>/editentry?agent-ref=STEVENS-NEWS&class-ref=Category&
caller=index&init=t
```

The edit parameter in that case asks OMAS to wait at the locate page form more information, rather than posting an empty form for creating a new object.

The difference with direct access is that edit pages contain a link back to the calling page here the application page (Fig.4).

8 Calling the Editor from a Web Dialog

When calling the web editor from a dialog page like the one shown Fig.5, one must build the proper subdialog task.

Assume for example that we want to add a new item to our set of news. We define the task create-info:

```
;;;=============================================================== CREATE INFO
;;; this is a test for invoking a web page

(deftask "NEWS: create info"
  :doc "Task for invoking the news editor"
  :performative :command
  :dialog _create-info-conversation
  :indexes ("create" .6 "info" .3)
  )
```

The create-info-conversation will only have a single input state.

Figure 4: An edit page with a link back to the calling page (bottom left)

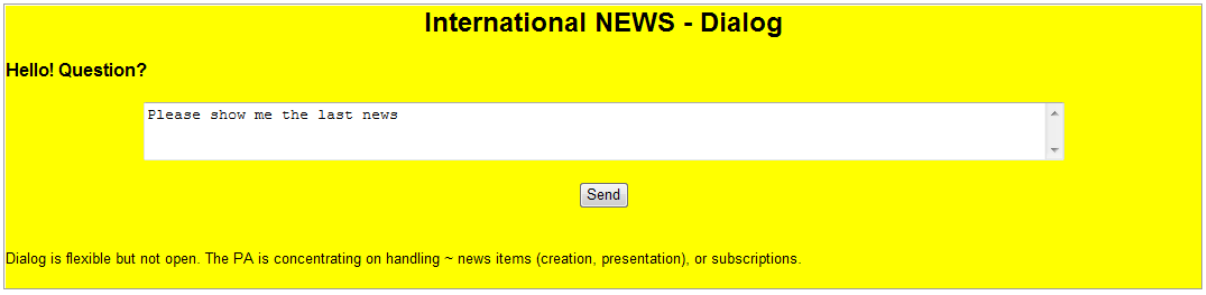

Figure 5: A web page containing a dialog window

```
1  ;;;=============================================================================
2  ;;;
3  ;;;                              NEWS: CREATE INFO
4  ;;;
5  ;;;=============================================================================
6  ;;; this conversation is a test for calling a web editor page
7
8  (defsubdialog _create-info-conversation
9    (:label "News: add keyword conversation")
10   (:explanation  "Master is adding a new keyword for indexing news items.")
11   (:states _create-info-conversation; required by defstate
12            _nci-entry-state
13            ))
14
15 ;;;------------------------------------------------------------------ (NCI) ENTRY-STATE
16 ;;; this is the entry and exit state. It sends back info to call the web editor
17
18 (defstate _nci-entry-state
19   (:entry-state _create-info-conversation)
20   (:label "NEWS: calling the editor to create an info")
21   (:explanation "Send the right parameters to the editor")
22   (:transitions
23   (:exec
24     (let ((index (moss::web-get-gate moss::conversation)))
25       ;; should set answer into *answer-list*
26       (moss::web-set-text moss::conversation '(:create "stevens-news" "news item"))
27       ;; clean up conversation
28       (web-clear-gate moss::conversation)
29       (web-clear-tag moss::conversation)
30       ;; and open gate to let PA return an answer to WEBNEWS
31       (omas::web-open-gate index)
32       ))
33   (:always :success))
34   ))
```

This is a standard conversation state, with and :exec option (line 23).
line 24: we recover the index of the synchronization gate used when running the dialog from a web browser
line 26: we set the text to send back to be the list, inserting it into the right place by using the omas::web-set-text function.

```
(:create "stevens-news" "news item")
```

which means that we want to create a new instance (individual) of news item and the owner is the agent stevens-news.
line 28 removes the gate index from the conversation object
line 29 removes the web tag from the conversation object
line 31 opens the gate, resuming the server process
line 33 ends the conversation turn by declaring it a success

One should be careful that this call to the editor does not interfere with a regular add-info subdialog.