



Easy Publisher Template API v1.6

- 1 Introduction2
- 2 Templates and Views2
 - 2.1 Easy Templates.....2
 - 2.2 Easy DTMLViews2
 - 2.3 Editable Regions3
 - 2.4 Properties.....3
 - 2.5 Images3
- 3 Handling paths and URLs4
 - 3.1 Virtual and physical paths and URLs.....4
 - 3.2 Path and URL attributes4
 - 3.3 Virtual site support methods5
- 4 Easy Navigation service7
 - 4.1 MenuList Format7
 - 4.2 MenuMap Format7
 - 4.3 Methods7
- 5 Easy News Service Object9
 - 5.1 Methods9
- 6 Easy Search Service Object10
 - 6.1 Properties.....10
 - 6.2 Methods10
- 7 Easy Mail Service object11
- 8 Easy Contact Form templates.....12
 - 8.1 Views12
 - 8.2 Methods13
 - 8.3 Advanced form rendering13
 - 8.4 Extending contact forms.....14



1 Introduction

This document covers the API for developing templates and sites with Easy Publisher 1.6. It does not cover HTML, DTML, Python or Zope. With the example code in this document, together with the example templates that are shipped with Easy Publisher, somebody with experience in HTML should be able to create templates. Used in this document is the tuple, list and dictionary collection types in Python, and the dtml-in, dtml-if, dtml-var and dtml-let tags from DTML. If you have no previous experience with Python or DTML it may help if you read up on these types and tags first.

For further information on HTML we recommend the W3C standards:
<http://www.w3.org/>

For further information about DTML and Zope we recommend the Zope Book:
<http://www.zope.org/Documentation/ZopeBook/>

For further information about the Python language:
<http://www.python.org/Documentation/>

2 Templates and Views

The Easy Publisher templates reside in the Easy Editor object. There are no restrictions to the HTML you can create in your view, so you have full freedom to create templates with any kind of design. Most of the templating you do are standard HTML-code, with some simple DTML.

2.1 Easy Templates

In the EasyEditor object you can create Easy Template objects. A template consists of at least one view and usually one or several editable regions. Typically the template also uses several shared DTML methods and images, and sometimes even Python scripts and SQL queries.

Each template is usually a type of document, such as a plain documents, a front page with a news box or a document with a picture box. You typically have different templates for different objects such as News Items and Contact Forms.

The view contains the HTML and DTML code that make up the page. If you have code that is shared by several views, place this code in DTML Methods in the template, and call it from the views with a dtml-var tag. If you have code that is shared by several templates, place this code in DTML methods in the Easy Editor object.

2.2 Easy DTMLViews

In the view you create the HTML and DTML code that make up a page. Since each template can have several views, and a complete webdesign often ends up with 3-10 templates, the total number of views often are in the 15-30 range. Of course you don't want to repeat the design code over and over in each of these, since a change would then force you to change all the views. Therefore the code of a typical view is often something like this:

```
<dtml-var header>
<div id="region"><dtml-var region missing=""></div>
<dtml-var footer>
```

As you see the actual design code is split up in two parts, a header and a footer, where the header contains all the code before the content area, and the footer all the code after. The header and footer object are DTMLMethod objects located in the EasyEditor object. Most of the time these are in turn split up in smaller code-pieces, also located in the EasyEditor object, where each piece contains a small and easy to use piece of code, such as the menu or the navigation trail.



2.3 Editable Regions

When creating editable regions you have a choice of how to store the data in the regions. For most uses, the DTMLStorage is the best option. This creates a DTML method in the document for each editable region. So, unless you have special demands, use the DTMLStorage.

To include an editable region in a view so that it is editable from the WYSIWYG editor, use a tag like this:

```
<div id="region"><dtml-var region missing=""></div>
```

Replace the occurrences of 'region' with the name of the region.

2.4 Properties

These are the properties that are set by Easy Document objects to select the template settings for the document. You can use these from the DTML code as needed:

viewid	Returns the id of the current View.
templateid	Returns the id of the current Template.
viewmode	Returns 'view' if page is viewed normally and 'edit' if the view is viewed in the Editor.
here	Returns the editable document that is being accessed in its original acquisition context.
this	The standard DTML variable "this" returns the view being used. Use if "this" in Easy Publisher templates is currently discouraged, as the behaviour may change in future versions of Easy Publisher.
browserfolder	returns a Browser Folder that correspond to the current User Agent or None if there isn't a Browser Folder that matches the User Agent.
template	Returns Template object.
editor	Returns EasyEditor object.
EASYEDITORURL	The URL to the current EasyEditor.
EASYNAVIGATIONURL	The URL to the current EasyNavigation.
SITEROOTURL	The URL to the current SiteRoot.

2.5 Images

It's always a good idea to separate the images used in the design from the code that makes up the design to reduce clutter. In EasyEditor, create a directory called "images" and place all the images used in the design there. That way you get the grouped with the other design, inside the Easy Editor object, but separated from any code.

The images should always be called with an absolute path, so that the web browsers cache can be used for the images. You can either use them in the standard HTML way:

```

```

You can also use DTML to display the image:

```
<dtml-var "EasyEditor.images['image.gif']">
```

Using DTML has the advantage that the height, width and alt settings are fetched from the object, so you don't have to update the code if you update the image. On the other hand, this is only practical if you make the HTML code by hand, since a graphical HTML editor won't support this syntax.



3 Handling paths and URLs

Each objects location can be specified with an URL or with a path. URLs are a text string where each object is separated with a slash. URLs can be either absolute or relative. An absolute URL starts in a slash, signifying the root object. All object URLs in Easy Publisher also end in a slash, to make it easy to append sub objects, views or methods to the URL.

A path contains the same information as an URL, but in the format of a tuple of string values. An absolute path starts with an empty string.

These URLs and paths all signify the same object:

An absolute URL: `'/products/electronics/radio/hk/D500/'`

The above object location as a path. Note the empty string in the beginning, making this an absolute path: `('', 'products', 'electronics', 'radio', 'hk', 'D500')`

A relative URL that is relative from the `'/products/electronics/'` object: `'radio/hk/D500/'`

The corresponding path: `('radio', 'hk', 'D500')`

3.1 Virtual and physical paths and URLs

If you are using virtual hosting to have several domain names to the same Easy Publisher server you also need to differ between virtual and physical locations. If you for example use virtual hosting to map the `www.torped.se` domain to the `'/www_torped_se/'` object you do not want to display that object in the URLs to the visitors of the site. The object `'/www_torped_se/products/easypublisher/'` should be shown to customers as `'http://www.torped.se/products/easypublisher/'`, and not `'http://www.torped.se/www_torped_se/products/easypublisher/'`. The physical location would include the `www.torped.se` part, and the virtual location would exclude it.

In most cases you should use virtual paths and URLs in HTML and physical paths and URLs in function calls.

3.2 Path and URL attributes

All Easy Publisher objects have these attributes and methods for getting current paths and URLs:

3.2.1 physicalPath

Holds the absolute physical path.

For example: `('', 'www_torped_se', 'products', 'easypublisher')`

3.2.2 physicalURL

Holds the absolute physical URL

For example: `'/www_torped_se/products/easypublisher/'`

3.2.3 virtualPath

Holds the absolute virtual path.

For example: `('', 'products', 'easypublisher')`

3.2.4 virtualURL

Holds the absolute virtual path.

For example: `'/products/easypublisher/'`



3.2.5 physicalParentPath

Holds the absolute physical path to the current objects parent object.
For example: ('', 'www_torped_se', 'products')

3.2.6 virtualParentPath

Holds the absolute physical path to the current objects parent object.
For example: ('', 'products')

3.2.7 getPhysicalPathMap()

Returns a list of mapping (dictionaries) objects describing all objects id, path and url in the current context from the root to the current object.

```
[
  {
    'id': id,
    'path': path,
    'url': url
  },
]
```

3.2.8 getVirtualPathMap()

As getPhysicalPathMap() but does not include objects above the virtual root

3.2.9 mergePaths(path1, path2)

Returns the logical join of two paths.

3.2.10 url2path(url)

Converts a url to a path

3.2.11 path2url(url)

Converts a path to a url

3.3 Virtual site support methods

If you use virtual hosting you have the following methods to support you. These all require that you have installed and configured our enhanced virtual host monster in the root with the name 'EasyVHM'. If this isn't done most of them will just return an empty string.

3.3.1 getDomainFromPath(path=None)

Returns the domain name used for the physical path given. If none is given, it will return the domain name for the object it is called from.

Usage example: <dtml-var "getDomainFromPath(ob.physicalPath)">

3.3.2 getSiteRootPath(path=None)

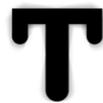
Returns the physical path to the virtual root object for the specified physical path. If no path is given, the path for the current object is used.

3.3.3 getSiteRootURL()

Returns the physical URL to the virtual root of the current object.

3.3.4 getSiteRoot()

Returns the virtual root object (usually an Easy Document).



3.3.5 splitPhysicalPath(path=None)

Splits the given physical paths and returns a tuple containing the physical path to the virtual root, and the objects virtual path.

3.3.6 isRoot()

Returns true if the current object is a Virtual Root

3.3.7 Conversion methods

These methods convert between virtual and physical paths and URLs:

virtualPathToPhysicalPath(path)

virtualPathToPhysicalURL(path)

virtualURLToPhysicalURL(url)

virtualURLToPhysicalPath(url)

physicalPathToVirtualPath(path)

physicalPathToVirtualURL(path)

physicalURLToVirtualURL(url)

physicalURLToVirtualPath(url)



4 Easy Navigation service

The EasyNavigation service object keeps track of all the navigation data in the system. You ask EasyNavigation for menus, and it will give you the menu data. This data can be in one of two formats, the MenuList format or the MenuMap format.

4.1 MenuList Format

The MenuList format is a simple list of 'brains', in the order they should be displayed. 'Brains' is the standard type of object that is returned from ZCatalog queries in Zope. These objects behave like if they were dictionaries and so are easy to use in python or in DTML.

A simple menu could be created like this

```
<dtml-in "EasyNavigation.getMenu(physicalPath)">
  <p><a class="menu" href="<dtml-var virtualURL>">
    <dtml-var navigation_title></a></p>
</dtml-in>
```

4.2 MenuMap Format

The MenuMap format is a hierarchical dictionary of menu information in the following layout:

```
{
  'id':           The objects Id,
  'title':        The objects Title,
  'navigation_title': The objects Navigation Title,
  'getVirtualPath': The objects Virtual Path,
  'absolute_url':  The objects URL,
  'brain':        The objects regular menu object (As returned by
getMenu),
  'submenu':      The submenu mapping or None,
  'is_current':   True|False,
  'in_path':      True|False,
}
```

4.3 Methods

4.3.1 getMenu(object_path, sort_on, sort_order)

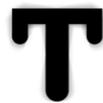
Returns a MenuList for the object physicalPath. You can set sort_on to any indexed field. If sort_on is not specified it will give you a list in the order it appears under the Navigation menu. Setting sort_order to 'reverse' will reverse the ordering.

Example: <dtml-in "EasyNavigation.getMenu(virtualPath, sort_order='reverse')">

4.3.2 getMenuMap(object_path, submenu)

Returns a MenuMap for the object object_path. submenu can be set to a number that specifies how many levels of submenus should be included in the result. If this is omitted or set to none only the top level will be included. If this is set to -1 then all levels will be included. If you set the object_path to (",) and submenu to -1 you will include all levels from the root down, and the result will therefore be a complete site map.

Example: <dtml-in "EasyNavigation.getMenuMap((" ,), submenu=-1)">



4.3.3 getNavigationContextMap(object_path, start_path)

Renders a MenuMap of the objects from start_path to object_path. This is similar to a getMenuMap, but the menus are expanded only for the root object, the current object and the objects inbetween. This is used for making expanding hierarchical menus like the one in the default Easy Publisher template.

Example: <dtml-let navmenu="EasyNavigation.getNavigationContextMap(physicalPath, ("),)">

4.3.4 getNavigationTrail(object_path)

Returns a MenuList containing the path of the object from the root down to the object.

Example: <dtml-in "Easynavigation.getNavigationTrail(virtualPath)



5 Easy News Service Object

The Easy News Service contains a list of all the News Items you have created in your site and is used to create listings of news items, such as press release listings and news archives.

5.1 Methods

5.1.1 `getNewsMenu(query, includepublished=1, includearchived=0)`

Returns a list of news entries for the news listing. Setting `includearchived` to 1 will mean that also archived news is included, and setting `includepublished` to 0 will mean that published (that is current) news items are included. The parameter `query` is a dictionary containing the query definition in standard Zope catalog syntax.

This example query will show all news items where the subject is 'Easy Publisher' that was published in 2001 :

```
getNewsMenu( {'subject':'Easy Publisher', 'effective_year': 2001},
             includepublished=1, includearchived=1)
```

5.1.2 `getYears()`

This method return all the years that news items exist for. This is useful if you want to make a news overview which have separate pages for separate years.

5.1.3 `getMonthsOfYear(text=None)`

Get used months in the news catalog as numbers if `text=None` or as text if `text=1`.

If there is a attribute 'monthnames' in the context it will use it for its month list. Monthnames should be a tuple or a list containing all earth months as text strings starting with January. You set monthnames most easily by creating a property in the root object where you list all the month names in the desired language.

If you don't have a 'monthnames' property the English monthnames will be returned.



6 Easy Search Service Object

Searching is done on the object catalog in the Easy Navigation service, where all searchable objects are indexed. The Easy Search Service is used to make it easier to do complex searches, and to extend your site with functionality such as keeping track of each sessions last search settings.

There can be several different EasySearchServices at once, but they then need to use different namespaces to avoid parameter conflicts.

6.1 Properties

6.1.1 Tracked Parameters

The Easy Search Service remembers tracked parameters until the current session times out. This means that they only needs to be set once, and will then be available in REQUEST for the remainder of the session. This is useful to keep track of a visitors last search, so they don't have to reenter it each time they go to the search page. This makes it possible for you to always show the last search results when a visitor goes to the search page. This means the visitor doesn't have to redo the search just because he has moved out of the search page.

When defining the parameters to be tracked you can optionally specify a parameter type by adding '<type>' after the parameter name and a default by adding a '=<default>'.

The following parameters are tracked by default:

```
query
query_start:int=1
batch_size:int=10
orphan:int
overlap:int
searchtext
sorton
pre_sorton
pre_sortorder
```

6.1.2 Namespace

EasySearchService can use namespaces, default namespace is set to EasySearchService. Namespace delimiter is __ (a double underscore). The namespace is added to the monitored parameter names to distinguish different search services parameters.

6.2 Methods

6.2.1 initSearch()

This method does the parameter tracking, and should be called from every page that uses the Easy Search Service.

6.2.2 getSearchResult()

Returns a list over search query matches. The search query will be created out of the parameters searchtext and meta_type. If the parameter advanced_search is set then the parameters search_for_categories and search_for_subjects will be added to the search query.

6.2.3 getPossibleItemsFor(name, additional_items=(), additional_values=(), exclude_values=(), cap=0, capword=0, underscore2space=0)

This method queries the catalog to see what values there are for an index. This is useful if you want a selection box in your search.



You can add additional values to the list, or additional items. The item list is created from the resulting values as an label-value pair, where `cap`, `capword` and `underscore2space` will dictate how the labels are formatted.

cap This makes the item values into all capital labels
capword This makes the item values have the first letter capitalized
underscore2space This converts any underscores in the values to spaces.

7 Easy Mail Service object

The Easy Mail Service object is an extension to the standard Zope object `MailHost`, and is used in the same way. It is currently only required by the Easy Notification service, but can be used by anybody that needs a `MailHost` object.

See the `MailHost` and `DTML-sendmail` documentation for further information.



8 Easy Contact Form templates

An Easy Contact Form object is used to easily create a form that can be filled in by visitors and the result sent to one (or several) form handlers, including the built-in sendmail handler that sends the form with e-mail.

Making on-line forms has traditionally been a complex task. You need to write a form handler script that verifies the form input, you need to take care of the problematic form standard that for example makes no difference between a checkbox that doesn't exist and one that isn't checked. You also need to tell your visitor exactly what the problem was if the verification failed, and which field that had the error. Some browsers also doesn't remember the values entered if you step backwards through the browser history, which means that if an error appear, you need to redisplay the form, with validation errors printed by the failing field. And finally, you need to create a script to perform an action based on the form input.

Easy Contact Form incorporates the Formulator product to validate the form input, so you don't have to do this yourself. Easy Publisher wraps and extends this functionality and together with the templating system and the default sendmail form handler, this means that it is possible to create mail forms with absolutely no programming at all. It also means that for any other type of form, the only thing the programmer need to worry about is taking care of the validated data that comes out of a correctly filled out form.

This means that creating forms is now a fast and simple job that requires little effort, and making those forms you have longed for is now within your reach.

See the Easy Contact Form How-To for more information on how to create and configure Easy Contact Forms.

8.1 Views

Easy Contact Form objects need to have specialized Templates. These templates need to have some specialized views; the viewview, the errorview and the thankview. A printview is also recommended.

8.1.1 viewview

The default view for Easy Contact Forms, renders the Form and additional Editable Regions. Typically you use two editable regions in this form, and include them in the view as follows:

```
<div id="over" style="width:100%;"><dtml-var over missing=""></div>
<dtml-var "Formulator.render()">
<div id="under" style="width:100%;"><dtml-var under missing=""></div>
```

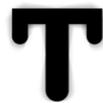
8.1.2 thankview

The view that shows a thank you message when the form was successfully completed. Can contain Editable Regions from the viewview as well as its own Editable Regions.

8.1.3 errorview

The view that shows an error message when the form couldn't be successfully send. Can contain Editable Regions from the viewview as well as its own Editable Regions.

Note that this view is not displayed when the validation of the form fails, but only when the sending of the mail fails or something else goes wrong. When the validation fails, the viewview is displayed again. The render method should be able to display the field validation errors (see Advanced Form rendering, below). This means that visitors doesn't have to step back to correct the form errors, but see the form and any error messaged directly, making form usage easier, more intuitive and browser safe.



8.1.4 printview

A regular printable version of the page, may contain values from the completed Form after the message has been sent. If you render the form it's a good idea to omit the send button or otherwise disable the form since it's only for printing and shouldn't work.

8.2 Methods

8.2.1 index_html(client, REQUEST, viewname)

Renders the view defined by viewname, defaults to viewview.

8.2.2 send(client, REQUEST)

Tries to send the form. If it succeeds it displays the thankview, if it fails it displays the errorview. If validation fails it displays the viewview again.

8.3 Advanced form rendering

8.3.1 Automatic rendering

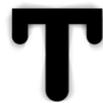
In most cases you do not want to write the HTML code that renders the form manually, since that means updating the HTML code as well as formulator each time you make changes. In that case you can use Formulator's own rendering methods. The method `get_fields()` will return a list of fields that you can iterate over, and the method `render_from_request()` can be used to make a default rendering of the form field.

```
<dtml-in "Formulator.get_fields()">
  <dtml-let field=sequence-item>
    <dtml-var "field.render_from_request(REQUEST)"><br>
  </dtml-let>
</dtml-in>
```

8.3.2 Displaying field validation errors

Formulator can validate the form fields in various ways, so that some fields must be entered, and email fields must contain valid email addresses. When form validation is used (which is almost always) you want to display the form errors. Each field that fails validation gets a variable called `field_error_{fieldname}` containing the error. Therefore you can render the error for a field with code like this:

```
<dtml-if "REQUEST.has_key('field_error_' + field.id)">
  <span class="required-text">
    <dtml-var "_['field_error_' + field.id]">
  </span>
</dtml-if>
```



8.3.3 Example render method

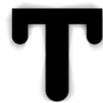
Here is an example of code that can be used to display a form that will show the validation errors if any. This code typically would be entered in a DTML method in the form which is called from the viewview.

```
<form action="&dtml-virtualURL;" method="post">
<table cellpadding="0" cellspacing="0">
  <dtml-in "Formulator.get_fields()">
    <dtml-let field=sequence-item>
      <tr>
        <td>
          <table cellpadding="0" cellspacing="0"><tr>
            <td valign="top">
              <dtml-if "field.is_required()">
                <span class="required-label">
                  <dtml-var "field.get_value('title')"></span>
                <dtml-if "not REQUEST.has_key(field.id)">
                  <span class="required-label">

                    <dtml-var required_sign missing="*"></span>
                </dtml-if>
              <dtml-else>
                <span class="optional-label">
                  <dtml-var "field.get_value('title')"></span>
              </dtml-if>
              <dtml-if "REQUEST.has_key('field_error_' + field.id)">
                <span class="required-text">
                  <dtml-var "_['field_error_' + field.id]"></span>
              </dtml-if>
            </td>
          </tr>
          <tr>
            <td class="optional-label">
              <dtml-var "field.render_from_request(REQUEST)">
            </td>
            <td valign="top">
          </td>
        </tr></table>
      </tr>
      <tr>
        <td colspan="2">
          
        </td>
      </tr>
    </dtml-let>
  </dtml-in>
  <tr>
    <td>
      <input type="submit" name="send:method" value="Send">
    </td>
  </tr>
</table>
</form>
```

8.4 Extending contact forms

You can extend contact form functionality in two ways. The first is the external validators you can write for Formulator, and the second is the form handler methods used to extend the functionality of the Contact Forms.



8.4.1 External validators

With external validation you can do any type of check on the fields. A validator is any kind of method that takes the parameters value and REQUEST. Value will contain the value of the field to be checked, and REQUEST will contain the complete REQUEST variable, including the other fields. An external validator should return 1 if the validation succeeds and 0 if it fails.

The easiest way to make validators is to create Python scripts with "value, REQUEST" as parameters. Here is an example of a script that makes sure that two passwords entered are the same:

```
return (REQUEST['field_password'] == REQUEST['field_password2']) and value
```

Here is an example of a script that looks up a username and a password from an SQL database, to validate the login:

```
userdata = context.validate_subscriber(userName=REQUEST['field_userName'],
    password=REQUEST['field_password'])
return len(userdata) == 1 #Wrong username or password
```

The validate_subscriber code is an SQL Method with the userName and password as parameters, and the following SQL code:

```
select * from Subscribers where <dtml-sqltest userName type="string"> and
<dtml-sqltest password type="string">
```

So as you see you can easily do any type of validation, including validating the entered data against an SQL database.

8.4.2 Form handlers

With Easy Publisher 1.6 form handlers enable you to use the Easy Contact Form as a front end for any kind of form you need, not only mail forms. A form handler is a method (typically a Python Script) with two parameters: result and REQUEST. Result is a dictionary of the form variables, and REQUEST is as usual the REQUEST variable.

A form handler should return the result variable if everything went well, and raise an exception if something goes wrong. If you don't return the result, the next handler will fail. This handing over of the result dictionary also means that a handler has the possibility to change the result, either by changing some of the variables, or adding new fields. This makes it possible to have a method that calculates a sum or fetches information from a database, which is then handed over to the sendmail handler so that the new data can be included in the mail sent.

You enter the name of the form handler under the "properties" page. By default an internal handler called "sendmail" is the only handler used. If you remove this handler no mail will be sent. If you have several handlers you should put the sendmail last, since sending a mail is nothing that can be undone, while most other actions you do in Zope, including writing to an SQL database, will be rolled back if something fails in a later handler.

Here is an example of a form handler that logs messages to a TinyTable called "messages".

```
from string import join

context.messages.setRow(
    email=result.get('email',''),
    name=result.get('name',''),
    message=result.get('message',''),
)

return result
```