Albatross Documentation

Release 1.40

Object Craft

March 15, 2010

CONTENTS

CHAPTER

ONE

FRONT MATTER

Abstract

The Albatross package is a toolkit which assists in the construction of web applications using the Python language.

See Also:

Python Language Web Site for information on the Python language

Albatross Users Mailing List for discussion regarding application development using Albatross

Albatross Web Site for information on the Albatross package

Albatross Users Wiki for user contributed information concerning Albatross

Apache Web Site for information on the Apache web server

Mod Python Web Site for information on mod_python

FastCGI Web Site for information on FastCGI

CHAPTER

TWO

THANK YOU

Object Craft wish to thank the following people.

Tim Churches For all of the time he has spent reviewing the documentation and pointing out errors.

Lewis Bergman For a constant stream of documentation fixes and suggestions

Gregory Bond For maintaining the FreeBSD port.

Matt Goodall For the FastCGI and (standard Python) BaseHTTPServer deployment code.

Fabian Fagerholm For building the Debian package.

CHAPTER

THREE

INTRODUCTION

Albatross is a small and flexible toolkit for developing highly stateful web applications. The toolkit is lightweight enough to use for CGI applications. It provides the following:

- Browser based sessions via automatically generated hidden form fields.
- Server side sessions via a session server or file based session store.
- Implicit form population and browser request handling.
- Powerful and extensible templating system which promotes separation of presentation and implementation for improved program maintainability.

  Pagination of sequences and tree browsing are handled implicitly in the templating system.

  Macros allow repeated HTML and special effects HTML to be defined in one location.

  Lookup tables translate internal program values to arbitrary template code.
- Applications can be deployed as either CGI programs or as `mod_python` module with minor changes to program mainline. Custom deployment can be achieved by developing your own `Request` class.
- Highly modular application framework which is flexible and extensible which allows many different application construction models. Many `Application` classes are provided and many more are possible.
- Comprehensive documentation including many installable samples.

A primary design goal of Albatross is that it be small and easy to use and extend. Most of the toolkit is constructed from a collection of mixin classes. You are encouraged to look at the code and to think of new ways to combine the Albatross mixin classes with your own classes.

Object Craft developed Albatross because there was nothing available with the same capabilities which they could use for consulting work. For this reason the toolkit is important to Object Craft and so is actively maintained and developed.

CHAPTER

FOUR

INSTALLATION

4.1 Prerequisites

Note that, where possible, you should install packages provided by your distribution, rather than building all the dependancies yourself.

- **Python** 2.3 or later.

  http://www.python.org/

If you wish to regenerate the documentation, you will also need:

- **Sphinx** 0.6.2 or later.

  http://sphinx.pocoo.org/

- **LaTeX**

  Only required if you want to build the PDF or PostScript documentation.

  On Ubuntu, install the `texlive` package (and dependancies).

- **dia**

  http://live.gnome.org/Dia

  The diagrams in the documentation are edited and rendered to PNG and EPS format using **dia** version 0.97.

- **GraphViz**

  http://www.graphviz.org/

  Used to auto-generate some diagrams.

4.2 Installing

Unpack the package archive to extract the package source:

```
tar xzf albatross-1.40.tar.gz
```

This will create an `albatross-1.40` subdirectory. Inside the `albatross-1.40` directory are a number of directories:

- **albatross**

  Contains the Python code for the package.

- **doc**

  Contains the documentation source.

- **samples**

Contains all of the sample programs discussed in this document.

- **session-server**

  Contains a simple session server which works with the Albatross server-side session mixin classes.

- **test**

  Contains the unit tests.

The `Albatross` package uses the `distutils` package so all you need to do is type the following command as root from the top level directory:

```
python setup.py install
```

If you have problems with this step, make sure that you contact the package author so that the installation process can be made more robust for other people.

If you already have a copy of Albatross installed and wish to test the new release before installing it globally then you can install an application private copy. If your application is installed in `/path/to/proj` then the following command will install a copy of Albatross that is only visible to that application:

```
python setup.py install --install-lib /path/to/proj --install-scripts /path/to/proj
```

If you wish to build the documentation, run:

```
cd doc
make pdf
```

Or:

```
cd doc
make html
```

4.3 Testing

There are a number of unit tests included in the distribution. These have been developed using the `unittest` package which is standard in Python 2.1 and later.

Run the following commands to perform the unit tests:

```
make test
```

If you want to do more than run the unit tests, you can start to work your way through the samples which are presented in this document.

CHAPTER

FIVE

TEMPLATES USER GUIDE

There are many different ways which you can use Albatross to assist in the construction of a web application. The purpose of this guide is to slowly introduce the features of Albatross to allow you to learn which features can be useful in your application.

All of the example programs in this chapter are distributed with the source in the `samples/templates` directory.

5.1 Introduction to CGI

This section presents a very simple program which will introduce you to CGI programming. This serves two purposes; it will verify that your web server is configured to run CGI programs, and it will demonstrate how simple CGI programming can be.

The sample program from this section is supplied in the `samples/templates/simple1` directory and can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/templates/simple1
python install.py
```

The `simple.py` program is show below.

```
#!/usr/bin/python

print 'Content-Type: text/html'
print
print '<html>'
print ' <head>'
print '  <title>My CGI application</title>'
print ' </head>'
print ' <body>'
print '  Hello from my simple CGI application!'
print ' </body>'
print '</html>'
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/simple1/simple.py.

If after installing the program locally you do not see a page displaying the text “Hello from my simple CGI application!” then you should look in the web server error log for clues. The location of the error log is specified by the `ErrorLog` directive in the Apache configuration. On a Debian Linux system the default location is `/var/log/apache/error.log`. While developing web applications you will become intimately familiar with this file.

Knowing how Apache treats a request for a document in the `cgi-bin` directory is the key to understanding how CGI programs work. Instead of sending the document text back to the browser Apache executes the “document” and sends the "document" output back to the browser. This simple mechanism allows you to write programs which generate HTML dynamically.

If you view the page source from the `simple1.py` application in your browser you will note that the first two lines of output produced by the program are not present. This is because they are not part of the document. The first line is an HTTP (Hypertext Transfer Protocol) header which tells the browser that the document content is HTML. The second line is blank which signals the end of HTTP headers and the beginning of the document.

You can build quite complex programs by taking the simple approach of embedding HTML within your application code. The problem with doing this is that program development and maintenance becomes a nightmare. The essential implementation (or business level) logic is lost within a sea of presentation logic. The impact of embedding HTML in your application can be reduced somewhat by using a package called `HTMLgen`. [1]

The other way to make complex web applications manageable is to separate the presentation layer from the application implementation via a templating system. This is also the Albatross way.

5.2 Your First Albatross Program

This section rewrites the sample CGI program from the previous section as an Albatross program. The program uses the Albatross templating system to generate the HTML document.

The sample program from this section is supplied in the `samples/templates/simple2` directory and can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/templates/simple2
python install.py
```

All of the HTML is moved into a template file called `simple.html`.

```html
<html>
 <head>
  <title>My CGI application</title>
 </head>
 <body>
  Hello from my second simple CGI application!
 </body>
</html>
```

The `simple.py` program is then rewritten as shown below.

```python
#!/usr/bin/python
from albatross import SimpleContext

ctx = SimpleContext('.')
templ = ctx.load_template('simple.html')
templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/simple2/simple.py.

This is probably the most simple application that you can write using Albatross. Let's analyse the program step-by-step.

This first line imports the Albatross package and places the `SimpleContext` into the global namespace.

[1] `HTMLgen` can be retrieved from http://starship.python.net/crew/friedrich/HTMLgen/html/main.html (this URL is currently broken - try via the Wayback Machine). On Debian or Ubuntu Linux you can install the `python-htmlgen` package.

```
from albatross import SimpleContext
```

Before we can use Albatross templates we must create an execution context which will be used to load and execute the template file. The Albatross `SimpleContext` object should be used in programs which directly load and execute template files. The `SimpleContext` constructor has a single argument which specifies a path to the directory from which template files will be loaded. Before Apache executes a CGI program it sets the current directory to the directory where that program is located. We have installed the template file in the same directory as the program, hence the path `'.'`.

```
ctx = SimpleContext('.')
```

Once we have an execution context we can load template files. The return value of the execution context `load_template()` method is a parsed template.

```
templ = ctx.load_template('simple.html')
```

Albatross templates are executed in two stages; the first stage parses the template and compiles the embedded Python expressions, the second actually executes the template.

To execute a template we call it's `to_html()` method passing an execution context. Albatross tags access application data and logic via the execution context. Since the template for the example application does not refer to any application functionality, we do not need to place anything into the context before executing the template.

```
templ.to_html(ctx)
```

Template file output is accumulated in the execution context.

Unless you use one of the Albatross application objects you need to output your own HTTP headers.

```
print 'Content-Type: text/html'
print
```

Finally, you must explicitly flush the context to force the HTML to be written to output. Buffering the output inside the context allows applications to trap and handle any exception which occurs while executing the template without any partial output leaking to the browser.

```
ctx.flush_content()
```

5.3 Introducing Albatross Tags

In the previous section we presented a simple Albatross program which loaded and executed a template file to generate HTML dynamically. In this section we will place some application data into the Albatross execution context so that the template file can display it.

To demonstrate how Albatross programs separate application and presentation logic we will look at a program which displays the CGI program environment. The sample program from this section is supplied in the `samples/templates/simple3` directory and can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/templates/simple3
python install.py
```

The CGI program `simple.py` is shown below.

```
#!/usr/bin/python
import os
from albatross import SimpleContext

ctx = SimpleContext('.')
templ = ctx.load_template('simple.html')

keys = os.environ.keys()
keys.sort()
ctx.locals.keys = keys
ctx.locals.environ = os.environ

templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

The following lines construct a sorted list of all defined environment variables. It makes the display a little nicer if the values are sorted.

```
keys = os.environ.keys()
keys.sort()
```

The Albatross execution context is constructed with an empty object in the `locals` member which is used as a conduit between the application and the toolkit. It is used as the local namespace for expressions evaluated in template files. To make the environment available to the template file we simply assign to an attribute using a name of our choosing which can then be referenced by the template file.

```
ctx.locals.keys = keys
ctx.locals.environ = os.environ
```

The `SimpleContext` constructor save a reference (in the `globals` member) to the global namespace of the execution context to the globals of the code which called the constructor.

Now the template file `simple.html`. Two Albatross tags are used to display the application data; `<al-for>` and `<al-value>`.

```
<html>
 <head>
  <title>The CGI environment</title>
 </head>
 <body>
  <table>
   <al-for iter="name" expr="keys">
    <tr>
     <td><al-value expr="name.value()"></td>
     <td><al-value expr="environ[name.value()]"></td>
    <tr>
   </al-for>
  </table>
 </body>
</html>
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/simple3/simple.py.

The `<al-for>` Albatross tag iterates over the list of environment variable names we placed in the `keys` value (`ctx.locals.keys`).

All template file content enclosed by the `<al-for>` tag is evaluated for each value in the sequence returned by evaluating the `expr` attribute. The `iter` attribute specifies the name of the iterator which is used to retrieve each successive value from the sequence. The toolkit places the iterator object in the `locals` member of the execution context. Be careful that you do not overwrite application values by using an iterator of the same name as an application value.

The `<al-value>` Albatross tag is used to retrieve values from the execution context. The `expr` attribute can contain any Python expression which can legally be passed to the Python `eval()` function when the *kind* argument is `"eval"`.

Deciding where to divide your application between implementation and presentation can be difficult at times. In the example above, we implemented some presentation logic in the program; we sorted the list of environment variables. Let's make a modification which removes that presentation logic from the application.

The `simple.py` application is shown below.

```
#!/usr/bin/python
import os
from albatross import SimpleContext

ctx = SimpleContext('.')
templ = ctx.load_template('simple.html')

ctx.locals.environ = os.environ

templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

Now look at the new `simple.html` template file. By using the Albatross `<al-exec>` tag we can prepare a sorted list of environment variable names for the `<al-for>` tag.

```
<html>
 <head>
  <title>The CGI environment</title>
 </head>
 <body>
  <table>
   <al-exec expr="keys = environ.keys(); keys.sort()">
   <al-for iter="name" expr="keys">
    <tr>
     <td><al-value expr="name.value()"></td>
     <td><al-value expr="environ[name.value()]"></td>
    <tr>
   </al-for>
  </table>
 </body>
</html>
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/simple4/simple.py.

The `<al-exec>` tag compiles the contents of the `expr` tag by passing `"exec"` as the *kind* argument. This means that you can include quite complex Python code in the attribute. Remember that we want to minimise the complexity of the entire application, not just the Python mainline. If you start placing application logic in the presentation layer, you will be back to having an unmaintainable mess.

Just for your information, the `<al-exec>` tag could have been written like this:

```
   <al-exec expr="
keys = environ.keys()
keys.sort()
">
```

5.3.1 Eliminating the Application

Let's revisit our first Albatross application with the `simple.py` sample program in the `samples/templates/simple5` directory.

```
#!/usr/bin/python
from albatross import SimpleContext

ctx = SimpleContext('.')
templ = ctx.load_template('simple.html')
templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

Now consider the template file `simple.html`.

```
<html>
 <head>
  <title>The CGI environment</title>
 </head>
 <body>
  <table>
   <al-exec expr="
import os
keys = os.environ.keys()
keys.sort()
">
   <al-for iter="name" expr="keys">
    <tr>
     <td><al-value expr="name.value()"></td>
     <td><al-value expr="os.environ[name.value()]"></td>
    <tr>
   </al-for>
  </table>
 </body>
</html>
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/simple5/simple.py.

You will notice that we have completely removed any application logic from the Python program. This is a cute trick for small example programs, but it is definitely a bad idea for any real application.

5.4 Building a Useful Application

In the previous section we saw how Albatross tags can be used to remove presentation logic from your application. In this section we will see how with a simple program we can serve up a tree of template files.

If you look at the output of the `simple4/simple.py` program you will notice the following lines:

```
REQUEST_URI     /cgi-bin/alsamp/simple4/simple.py
SCRIPT_FILENAME /usr/lib/cgi-bin/alsamp/simple4/simple.py
SCRIPT_NAME     /cgi-bin/alsamp/simple4/simple.py
```

Now watch what happens when you start appending extra path elements to the end of the URL. Try requesting the following with your browser: http://www.object-craft.com.au/cgi-bin/alsamp/simple4/simple.py/main.html.

You should see the following three lines:

```
REQUEST_URI     /cgi-bin/alsamp/simple4/simple.py/main.html
SCRIPT_FILENAME /usr/lib/cgi-bin/alsamp/simple4/simple.py
SCRIPT_NAME     /cgi-bin/alsamp/simple4/simple.py
```

The interesting thing is that Apache is still using the `simple4/simple.py` program to process the browser request. We can use the value of the `REQUEST_URI` environment variable to locate a template file which will be displayed.

The sample application in the `samples/templates/content1` directory demonstrates serving dynamic content based upon the requested URI. The program can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/templates/content1
python install.py
```

The CGI program `content.py` is shown below.

```
#!/usr/bin/python
import os
from albatross import SimpleContext, TemplateLoadError

script_name = os.environ['SCRIPT_NAME']
request_uri = os.environ['REQUEST_URI']
page = request_uri[len(script_name) + 1:]
if not page or os.path.dirname(page):
    page = 'main.html'

ctx = SimpleContext('templ')
ctx.locals.page = page
try:
    templ = ctx.load_template(page)
except TemplateLoadError:
    templ = ctx.load_template('oops.html')

templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

To demonstrate this application we have three template files; `main.html`, `oops.html`, and `other.html`.

First `main.html`.

```
<html>
 <head>
  <title>Simple Content Management - main page.</title>
 </head>
 <body>
  <h1>Simple Content Management - main page</h1>
  <hr noshade>
  This is the main page.
 </body>
</html>
```

Now `other.html`.

```
<html>
 <head>
  <title>Simple Content Management - other page.</title>
 </head>
```

```
    <body>
     <h1>Simple Content Management - other page</h1>
     <hr noshade>
     This is the other page.
    </body>
   </html>
```

And finally the page for displaying errors; `oops.html`.

```
   <html>
    <head>
     <title>Simple Content Management - error page.</title>
    </head>
    <body>
     <h1>Simple Content Management - error page</h1>
     <hr noshade>
     <al-if expr="page == 'oops.html'">
      You actually requested the error page!
     <al-else>
      Sorry, the page <font color="red"><al-value expr="page"></font>
      does not exist.
     </al-if>
    </body>
   </html>
```

Test the program by trying a few requests with your browser:

- http://www.object-craft.com.au/cgi-bin/alsamp/content1/content.py
- http://www.object-craft.com.au/cgi-bin/alsamp/content1/content.py/main.html
- http://www.object-craft.com.au/cgi-bin/alsamp/content1/content.py/other.html
- http://www.object-craft.com.au/cgi-bin/alsamp/content1/content.py/error.html
- http://www.object-craft.com.au/cgi-bin/alsamp/content1/content.py/oops.html

Let's analyse the program step-by-step. The preamble imports the modules we are going to use.

```
#!/usr/bin/python
import os
from albatross import SimpleContext, TemplateLoadError
```

The next part of the program removes the prefix in the `SCRIPT_NAME` variable from the value in the `REQUEST_URI` variable. When removing the script name we add one to the length to ensure that the `"/"` path separator between the script and page is also removed. This is important because the execution context `load_template()` method uses `os.path.join()` to construct a script filename by combining the *base_dir* specified in the constructor and the name passed to the `load_template()` method. If any of the path components being joined begin with a `"/"` then `os.path.join()` creates an absolute path beginning at the `"/"`.

If no page was specified in the browser request then we use the default page `main.html`.

```
script_name = os.environ['SCRIPT_NAME']
request_uri = os.environ['REQUEST_URI']
page = request_uri[len(script_name) + 1:]
if not page:
    page = 'main.html'
```

The next section of code creates the Albatross execution context and places the requested filename into the `page` local attribute. It then attempts to load the requested file. If the template file does not exist the `load_template()` will raise a `TemplateLoadError` exception. We handle this by loading the error page `oops.html`.

The error page displays a message which explains that the requested page (saved in the page variable) does not exist.

```
ctx = SimpleContext('templ')
ctx.locals.page = page
try:
    templ = ctx.load_template(page)
except TemplateLoadError:
    templ = ctx.load_template('oops.html')
```

Looking at the error page oops.html, you will see a new Albatross tag <al-if>.

```
<al-if expr="page == 'oops.html'">
 You actually requested the error page!
<al-else>
 Sorry, the page <font color="red"><al-value expr="page"></font>
 does not exist.
</al-if>
```

The <al-if> tag allows you to conditionally include or exclude template content by testing the result of an expression. Remember that we placed the name of the requested page into the page variable, so we are able to display different content when the browser actually requests oops.html.

Finally, the remainder of the program displays the selected HTML page.

```
templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

5.5 Albatross Macros

In the previous section we demonstrated a program which can be used to display pages from a collection of template files. You might recall that the HTML in the template files was very repetitive. In this section you will see how Albatross macros can be used to introduce a common look to all HTML pages.

If we look at the main.html template file again you will notice that there really is very little content which is unique to this page.

```
<html>
 <head>
  <title>Simple Content Management - main page.</title>
 </head>
 <body>
  <h1>Simple Content Management - main page</h1>
  <hr noshade>
  This is the main page.
 </body>
</html>
```

Using Albatross macros we can place all of the boilerplate into a macro. Once defined, the macro can be reused in all template files.

The sample program from this section is supplied in the samples/templates/content2 directory and can be installed in your web server cgi-bin directory by running the following commands.

```
cd samples/templates/content2
python install.py
```

First consider the macro in the macros.html template file.

```
<al-macro name="doc">
<html>
 <head>
  <title>Simple Content Management - <al-usearg name="title"></title>
 </head>
 <body>
  <h1>Simple Content Management - <al-usearg name="title"></h1>
  <hr noshade>
  <al-usearg>
 </body>
</html>
</al-macro>
```

Now we can change main.html to use the macro.

```
<al-expand name="doc">
  <al-setarg name="title">main page</al-setarg>
  This is the main page.
</al-expand>
```

Likewise, the other.html file.

```
<al-expand name="doc">
  <al-setarg name="title">other page</al-setarg>
  This is the other page.
</al-expand>
```

And finally the error page oops1.html.

```
<al-expand name="doc">
  <al-setarg name="title">error page</al-setarg>
  <al-if expr="page == 'oops.html'">
   You actually requested the error page!
  <al-else>
   Sorry, the page <font color="red"><al-value expr="page"></font>
   does not exist.
  </al-if>
</al-expand>
```

We also have to modify the application to load the macro definition before loading the requested pages.

```
#!/usr/bin/python
import os
from albatross import SimpleContext, TemplateLoadError

script_name = os.environ['SCRIPT_NAME']
request_uri = os.environ['REQUEST_URI']
page = request_uri[len(script_name) + 1:]
if not page or os.path.dirname(page):
    page = 'main.html'

ctx = SimpleContext('templ')
ctx.load_template('macros.html').to_html(ctx)
ctx.locals.page = page
try:
    templ = ctx.load_template(page)
except TemplateLoadError:
    templ = ctx.load_template('oops.html')

templ.to_html(ctx)
```

```
print 'Content-Type: text/html'
print
ctx.flush_content()
```

Test the program by trying a few requests with your browser:

- http://www.object-craft.com.au/cgi-bin/alsamp/content2/content.py
- http://www.object-craft.com.au/cgi-bin/alsamp/content2/content.py/main.html
- http://www.object-craft.com.au/cgi-bin/alsamp/content2/content.py/other.html
- http://www.object-craft.com.au/cgi-bin/alsamp/content2/content.py/error.html
- http://www.object-craft.com.au/cgi-bin/alsamp/content2/content.py/oops.html

The only new line in this program is the following:

```
ctx.load_template('macros.html').to_html(ctx)
```

This loads the file which contains the macro definition and then executes it. Executing the macro definition registers the macro with the execution context, it does not produce any output. This means that once defined the macro is available to all templates interpreted by the execution context. The template is not needed once the macro has been registered so we can discard the template file.

When you use Albatross application objects the macro definition is registered in the application object so can be defined once and then used with all execution contexts.

There is one small problem with the program. What happens if the browser requests `macros.html`? Suffice to say, you do not get much useful output. The way to handle this problem is to modify the program to treat requests for `macros.html` as an error.

Let's revisit the macro definition in `macros.html` and see how it works. Albatross macros use four tags; `<al-macro>`, `<al-usearg>`, `<al-expand>`, and `<al-setarg>`.

```
<al-macro name="doc">
<html>
 <head>
  <title>Simple Content Management - <al-usearg name="title"></title>
 </head>
 <body>
  <h1>Simple Content Management - <al-usearg name="title"></h1>
  <hr noshade>
  <al-usearg>
 </body>
</html>
</al-macro>
```

The `<al-macro>` tag is used to define a named macro. The `name` attribute uniquely identifies the macro within the execution context. In our template file we have defined a macro called `"doc"`. All content enclosed in the `<al-macro>` tag will be substituted when the macro is expanded via the `<al-expand>` tag.

In all but the most simple macros you will want to pass some arguments to the macro. The place where the arguments will be expanded is controlled via the `<al-usearg>` tag in the macro definition. All macros accept an "unnamed" argument which captures all of the content within the `<al-expand>` tag not enclosed by `<al-setarg>` tags. The unnamed argument is retrieved within the macro definition by using `<al-usearg>` without specifying a `name` attribute.

In our example we used a fairly complex macro. If you are still a bit confused the following sections should hopefully clear up that confusion.

5.5.1 Zero Argument Macros

The most simple macro is a macro which does not accept any arguments. You might define the location of the company logo within a zero argument macro.

```
<al-macro name="small-logo">
<img src="http://images.company.com/logo/small.png">
</al-macro>
```

Then whenever you need to display the logo all you need to do is expand the macro.

```
<al-expand name="small-logo"/>
```

This allows you to define the location and name of your company logo in one place.

5.5.2 Single Argument Macros

The single argument macro is almost as simple as the zero argument macro. You should always use the unnamed argument to pass content to a single argument macro.

If you look at news sites such as http://slashdot.org/ you will note that they make heavy use of HTML tricks to improve the presentation of their pages. Single argument macros can be extremely useful in simplifying your template files by moving the complicated HTML tricks into a separate macro definition file.

Let's look at the top story at http://slashdot.org/ to illustrate the point. The title bar for the story is constructed with the following HTML (reformatted so it will fit on the page).

```
<table width="100%" cellpadding=0 cellspacing=0 border=0>
<tr>
<td valign=top bgcolor="#006666">
<img src="http://images.slashdot.org/slc.gif" width=13 height=16 alt="" align=top>
<font size=4 color="#FFFFFF" face="arial,helvetica">
<b>Another Nasty Outlook Virus Strikes</b>
</font>
</td>
</tr>
</table>
```

As you can see, most of the HTML is dedicated to achieving a certain effect. If you were using Albatross to construct the same HTML you would probably create a macro called `story-title` like this:

```
<al-macro name="story-title">
<table width="100%" cellpadding=0 cellspacing=0 border=0>
<tr>
<td valign=top bgcolor="#006666">
<img src="http://images.slashdot.org/slc.gif" width=13 height=16 alt="" align=top>
<font size=4 color="#FFFFFF" face="arial,helvetica">
<b><al-usearg></b>
</font>
</td>
</tr>
</table>
</al-macro>
```

Then you could generate the story title HTML like this:

```
<al-expand name="story-title">Another Nasty Outlook Virus Strikes</al-expand>
```

Since stories are likely to be generated from some sort of database it is more likely that you would use something like this:

```
<al-expand name="story-title"><al-value expr="story.title"></al-expand>
```

5.5.3 Multiple Argument Macros

Multiple argument macros are effectively the same as single argument macros that accept additional named arguments.

The following example shows a macro that defines multiple arguments and some template to expand the macro.

```
<al-macro name="multi-arg">
arg1 is "<al-usearg name="arg1">" and
arg2 is "<al-usearg name="arg2">" and
the default argument is "<al-usearg>".
</al-macro>

<al-expand name="multi-arg">
This is <al-setarg name="arg2">arg2 content</al-setarg>
the <al-setarg name="arg1">arg1 content</al-setarg>
default argument</al-expand>
```

When the above template is executed the following output is produced.

```
arg1 is "arg1 content" and
arg2 is "arg2 content" and
the default argument is "This is the default argument".
```

5.5.4 Nesting Macros

Let's revisit the http://slashdot.org/ HTML for a story and see how to use macros to assist in formatting the entire story summary.

Consider the rest of the story summary minus the header (reformatted to fit on the page):

```
<a HREF="http://slashdot.org/search.pl?topic=microsoft">
<img SRC="http://images.slashdot.org/topics/topicms.gif" WIDTH="75" HEIGHT="55"
     BORDER="0" ALIGN="RIGHT" HSPACE="20" VSPACE="10" ALT="Microsoft">
</a>
<b>Posted by <a HREF="http://www.monkey.org/~timothy">timothy</a>
 on  Sunday July 22, @11:32PM</b><br>
<font size=2><b>from the hide-the-children-get-the-gun dept.</b></font><br>
Goldberg's Pants writes: <i>
"<a HREF="http://www.zdnet.com/zdnn/stories/news/0,4586,2792260,00.html?chkpt=zdnnp1tp02">ZDNet</
and <a HREF="http://www.wired.com/news/technology/0,1282,45427,00.html">Wired</a>
are both reporting on a new virus that spreads via Outlook. Nothing
particularly original there, except this virus is pretty unique both
in how it operates, and what it does, such as emailing random
documents from your harddrive to people in your address book, and
hiding itself in the recycle bin which is rarely checked by virus
scanners."</i> I talked by phone with a user whose machine seemed
determined to send me many megabytes of this virus 206k at a time; he
was surprised to find that his machine was infected, as most people
probably would be. The anti-virus makers have patches, if you are
running an operating system which needs them.
```

The first task is to simplify is the topic specific image. There are a finite number of topics, and the set of topics does not change much over time. We could make effective use of the Albatross `<al-lookup>` tag to simplify this (`<al-lookup>` is discussed in section *Lookup Tables*):

```
<al-lookup name="story-topic">
 <al-item expr="'microsoft'">
  <a HREF="http://slashdot.org/search.pl?topic=microsoft">
   <img SRC="http://images.slashdot.org/topics/topicms.gif" WIDTH="75"
        HEIGHT="55" BORDER="0" ALIGN="RIGHT" HSPACE="20" VSPACE="10"
        ALT="Microsoft">
  </a>
 </al-item>
 <al-item expr="'news'">
   :
 </al-item>
</al-lookup>
```

Then to display the HTML for the story topic all we would need to do is the following:

```
<al-value expr="story.topic" lookup="story-topic">
```

Next we will simplify the acknowledgement segment:

```
<b>Posted by <al-value expr="story.poster" noescape>
 on <al-value expr="story.date" date="%A %B %d, @%I:%M%p"></b><br>
<font size=2><b>from the <al-value expr="story.dept"> dept.</b></font><br>
```

Finally we can bring all of these fragments together like this:

```
<al-macro name="story-summary">
 <al-expand name="story-title"><al-value name="story.title"></al-expand>
 <al-value expr="story.topic" lookup="story-topic">
 <b>Posted by <al-value expr="story.poster">
 on <al-value expr="story.date" date="%A %B %d, @%I:%M%p"></b><br>
 <font size=2><b>from the <al-value expr="story.dept"> dept.</b></font><br>
 <al-value expr="story.summary" noescape>
</al-macro>
```

Having defined the macro for formatting a story summary we can format a list of story summaries like this:

```
<al-for iter="i" expr="summary_list">
 <al-exec expr="story = i.value()">
 <al-expand name="story-summary"/>
</al-for>
```

Notice that all of the macros are referring directly to the `story` object which contains all of the data which pertains to one story.

If you find that your macros are referring to application data by name then you should consider using a function instead. Once functions have been implemented you might be able to use them as well as consider them.

5.6 Lookup Tables

The example macro used in the previous section introduced a new Albatross tag called `<al-lookup>`. In this section we will look at the tag in more detail.

The `<al-lookup>` tag provides a mechanism for translating internal program values into HTML for display. This is another way which Albatross allows you to avoid placing presentation logic in your application.

In a hypothetical bug tracking system we have developed we need to display information about bugs recorded in the system. The severity of a bug is defined by a collection of symbols defined in the `btsvalues` module.

```
TRIVIAL = 0
MINOR = 1
NORMAL = 2
MAJOR = 3
CRITICAL = 4
```

While the integer severity levels are OK for use as internal program values they are not very useful as displayed values. The obvious way to display a bug severity would be via the `<al-value>` tag.

```
Severity: <al-value expr="bug.severity">
```

Unfortunately, this would yield results like this:

```
Severity: 1
```

By using the `lookup` attribute of the `<al-value>` tag we are able to use the internal value as an index into a lookup table. The corresponding entry from the lookup table is displayed instead of the index.

The following is a table which translates the internal program value into HTML for display.

```
<al-lookup name="bug-severity">
 <al-item expr="btsvalues.TRIVIAL"><font color="green">Trivial</font></al-item>
 <al-item expr="btsvalues.MINOR">Minor</al-item>
 <al-item expr="btsvalues.NORMAL">Normal</al-item>
 <al-item expr="btsvalues.MAJOR"><font color="red">Major</font></al-item>
 <al-item expr="btsvalues.CRITICAL"><font color="red"><b>Critical</b></font></al-item>
</al-lookup>
```

The `btsvalues` module must be visible when the `<al-lookup>` tag is executed. You can place the `btsvalues` module in the the global namespace of the execution context by importing the `btsvalues` module in the same module which creates the `SimpleContext` execution context. When using other Albatross execution contexts you would need to import `btsvalues` in the module which called `run_template()` or `run_template_once()` to execute the `<al-lookup>` tag.

We invoke the lookup table by using the `lookup` attribute of the `<al-value>` tag.

```
Severity: <al-value expr="bug.severity" lookup="bug-severity">
```

Note that the `btsvalues` module does not need to be in the namespace at this point. The `expr` attributes in the `<a-item>` tags are evaluated once when the `<al-lookup>` tag is executed.

The `<al-lookup>` tag has the same runtime properties as the `<al-macro>` tag. You have execute the tag to register the lookup table with the execution context. Once the lookup table has been registered it is available to all template files executed in the same execution context.

When using Albatross application objects the lookup table is registered in the application object so can be defined once and then used with all execution contexts.

Each entry in the lookup table is enclosed in a `<al-item>` tag. The `expr` attribute of the `<al-item>` tag defines the expression which will be evaluated to determine the item's table index. As explained above, the expression is evaluated when the lookup table is executed, not when the table is loaded, or looked up (with the rare exception of a lookup being used earlier in the same template file that it is defined).

It is important to note that the content enclosed by the `<al-item>` tag is executed when the item is retrieved via an `<al-value>` tag. This allows you to place Albatross tags inside the lookup table that are designed to be evaluated when the table is accessed.

Finally, any content not enclosed by an `<al-item>` tag will be returned as the result of a failed table lookup.

5.7 White Space Removal in Albatross

If you were paying close attention to the results of expanding the macros we created in section *Albatross Macros* you would have noticed that nearly all evidence of the Albatross tags has disappeared. It is quite obvious that the Albatross tags are no longer present. A little less obvious is removal of whitespace following the Albatross tags.

Let's have a look at the `"doc"` macro again.

```
<al-macro name="doc">
<html>
 <head>
  <title>Simple Content Management - <al-usearg name="title"></title>
 </head>
 <body>
  <h1>Simple Content Management - <al-usearg name="title"></h1>
  <hr noshade>
  <al-usearg>
 </body>
</html>
</al-macro>
```

We can get a capture the result of expanding the macro by firing up the Python interpreter to manually exercise the macro.

```
>>> import albatross
>>> text = '''<al-macro name="doc">
... <html>
...  <head>
...   <title>Simple Content Management - <al-usearg name="title"></title>
...  </head>
...  <body>
...   <h1>Simple Content Management - <al-usearg name="title"></h1>
...   <hr noshade>
...   <al-usearg>
...  </body>
... </html>
... </al-macro>
... '''
>>> ctx = albatross.SimpleContext('.')
>>> templ = albatross.Template(ctx, '<magic>', text)
>>> templ.to_html(ctx)
>>> text = '''<al-expand name="doc">
... <al-setarg name="title">hello</al-setarg>
... </al-expand>
... '''
>>> expand = albatross.Template(ctx, '<magic>', text)
>>> ctx.push_content_trap()
>>> expand.to_html(ctx)
>>> result = ctx.pop_content_trap()
>>> print result
<html>
 <head>
  <title>Simple Content Management - hello</title>
 </head>
 <body>
  <h1>Simple Content Management - hello</h1>
  <hr noshade>
  </body>
</html>
```

Not only have the `<al-macro>` and `<al-expand>` tags been removed, the whitespace that follows those tags has also been removed. By default Albatross removes all whitespace following an Albatross tag that begins with a newline. This behaviour should be familiar to anyone who has used PHP.

Looking further into the result you will note that the `</body>` tag is aligned with the `<hr noshade>` tag above it. This is the result of performing the `<al-usearg>` substitution (which had no content) and removing all whitespace following the `<al-usearg>` tag.

This whitespace removal nearly always produces the desired result, though it can be a real problem at times.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.title = 'Mr.'
>>> ctx.locals.fname = 'Harry'
>>> ctx.locals.lname = 'Tuttle'
>>> templ = albatross.Template(ctx, '<magic>', '''<al-value expr="title">
...  <al-value expr="fname">
...  <al-value expr="lname">
... ''')
>>> ctx.push_content_trap()
>>> templ.to_html(ctx)
>>> ctx.pop_content_trap()
'Mr.HarryTuttle'
```

The whitespace removal has definitely produced an undesirable result.

You can always get around the problem by joining all of the `<al-value>` tags together on a single line. Remember that the whitespace removal only kicks in if the whitespace begins with a newline character. For our example this would be a reasonable solution.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.title = 'Mr.'
>>> ctx.locals.fname = 'Harry'
>>> ctx.locals.lname = 'Tuttle'
>>> templ = albatross.Template(ctx, '<magic>', '''<al-value expr="title"> <al-value expr="fna
>>> ctx.push_content_trap()
>>> templ.to_html(ctx)
>>> ctx.pop_content_trap()
'Mr. Harry Tuttle'
```

The other way to defeat the whitespace removal while keeping each `<al-value>` tag on a separate line would be to place a single trailing space at the end of each line. This would be a very bad idea because the next person to modify the file might remove the space without realising how important it was.

Note that there are trailing spaces at the end of each line in the `text` assignment. This should give you a clue about how bad this technique is.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.title = 'Mr.'
>>> ctx.locals.fname = 'Harry'
>>> ctx.locals.lname = 'Tuttle'
>>> templ = albatross.Template(ctx, '<magic>', '''<al-value expr="title">
... <al-value expr="fname">
... <al-value expr="lname">
... ''')
>>> ctx.push_content_trap()
>>> templ.to_html(ctx)
>>> ctx.pop_content_trap()
'Mr. \nHarry \nTuttle'
```

A much better way to solve the problem is to explicitly tell the Albatross parser that you want it to do something different with the whitespace that follows the first two `<al-value>` tags.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.title = 'Mr.'
>>> ctx.locals.fname = 'Harry'
>>> ctx.locals.lname = 'Tuttle'
>>> templ = albatross.Template(ctx, '<magic>', '''<al-value expr="title" whitespace="indent">
...  <al-value expr="fname" whitespace="indent">
...  <al-value expr="lname">
... ''')
>>> ctx.push_content_trap()
>>> templ.to_html(ctx)
>>> ctx.pop_content_trap()
'Mr. Harry Tuttle'
```

The above variation has told the Albatross interpreter to only strip the trailing newline, leaving intact the indent on the following line. The following table describes all of possible values for the `whitespace` attribute.

Value	Meaning
`'all'`	Keep all following whitespace.
`'strip'`	Remove all whitespace - this is the default.
`'indent'`	Keep indent on following line.
`'newline'`	Remove all whitespace and substitute a newline.

Note that when the trailing whitespace does not begin with a newline the `'strip'` and `'indent'` whitespace directives are treated exactly like `'all'`.

5.8 Using Forms to Receive User Input

Nearly all web applications need to accept user input. User input is captured by using forms. We will begin by demonstrating the traditional approach to handling forms, then in later sections you will see how Albatross can be used to eliminate the tedious shuffling of application values in and out of form elements.

Let's start with a program that presents a form to the user and displays to user response to the form. The sample program from this section is supplied in the `samples/templates/form1` directory and can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/templates/form1
python install.py
```

The CGI program `form.py` is shown below.

```
#!/usr/bin/python
import cgi
from albatross import SimpleContext

ctx = SimpleContext('.')
ctx.locals.form = cgi.FieldStorage()

templ = ctx.load_template('form.html')
templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

There are no surprises here, we are using the standard Python `cgi` module to capture the browser request. We want to display the contents of the request so it is placed into the execution context.

The `form.html` template file is used to display present a form and display the browser request.

```
<html>
 <head>
  <title>Display Form Input</title>
 </head>
 <body>
  Input some values to the following form and press the submit button.
  <form method="post" action="form.py">
   Text field: <input name="text"><br>
   Singleton checkbox: <input type="checkbox" name="singleton"><br>
   Checkbox group:
   <input type="checkbox" name="group" value="check1">
   <input type="checkbox" name="group" value="check2"><br>
   Radio buttons:
   <input type="radio" name="radio" value="radio1">
   <input type="radio" name="radio" value="radio2"><br>
   Option menu: <select name="select"><option>option1<option>option2<option>option3</select>
   <input type="submit" value="submit">
  </form>
  <al-include name="form-display.html">
 </body>
</html>
```

We have placed the form display logic in a separate template file because we wish to reuse that particularly nasty piece of template. The form display template is contained in `form-display.html`.

If you do not understand how the `FieldStorage` class from the `cgi` module works, do not try to understand the following template. Refer to section *Using Albatross Input Tags* which contains a small explanation of the `FieldStorage` class and some Python code that performs the same task as the template.

```
<al-for iter="f" expr="form.keys()">
 <al-exec expr="field = form[f.value()]">
 <al-if expr="type(field) is type([])">
  Field <al-value expr="f.value()"> is list:
  <al-for iter="e" expr="field">
   <al-exec expr="elem = e.value()">
   <al-if expr="e.index() > 0">, </al-if>
   <al-value expr="elem.value">
  </al-for>
 <al-else>
  Field <al-value expr="f.value()"> has a single value:
  <al-value expr="field.value">
 </al-if>
 <br>
</al-for>
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/form1/form.py.

You will notice that each time you submit the page it comes back with all of the fields cleared again.

Typically web applications that generate HTML dynamically will hand construct `<input>` tags and place application values into the `value` attributes of the input tags. Since we are using Albatross templates we do not have the ability to construct tags on the fly without doing some very nasty tricks. Fortunately Albatross supplies some tags that we can use in place of the standard HTML `<input>` tags.

5.9 Using Albatross Input Tags

In the previous section we saw how web applications can capture user input from browser requests. This section explains how Albatross `<al-input>` tags can be used to take values from the execution context and format them as `value` attributes in the HTML `<input>` tags sent to the browser.

The sample program from this section is supplied in the `samples/templates/form2` directory and can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/templates/form2
python install.py
```

The first change is in the `form.html` template file.

```
<html>
 <head>
  <title>Display Form Input</title>
 </head>
 <body>
  Input some values to the following form and press the submit button.
  <form method="post" action="form.py">
   Text field: <al-input name="text"><br>
   Singleton checkbox: <al-input type="checkbox" name="singleton"><br>
   Checkbox group:
   <al-input type="checkbox" name="group" value="check1">
   <al-input type="checkbox" name="group" value="check2"><br>
   Radio buttons:
   <al-input type="radio" name="radio" value="radio1">
   <al-input type="radio" name="radio" value="radio2"><br>
   Option menu:
   <al-select name="select">
    <al-option>option1</al-option>
    <al-option>option2</al-option>
    <al-option>option3</al-option>
   </al-select>
   <al-input type="submit" name="submit" value="Submit">
  </form>
  <al-include name="form-display.html">
 </body>
</html>
```

We need to place some values into the execution context so that the Albatross `<al-input>` tags can display them. The easiest thing to do is to place the browser submitted values into the execution context.

The documentation for the Python `cgi` module is quite good so I will not try to explain the complete behaviour of the `FieldStorage` class. The only behaviour that we need to be aware of for our program is what it does when it receives more than one value for the same field name.

The `FieldStorage` object that captures browser requests behaves like a dictionary that is indexed by field name. When the browser sends a single value for a field, the dictionary lookup yields an object containing the field value in the `value` member. When the browser sends more than one value for a field, the dictionary lookup returns a list of the objects used to represent a single field value.

Using this knowledge, the `form.py` program can be modified to merge the browser request into the execution context.

```
#!/usr/bin/python
import cgi
from albatross import SimpleContext

form = cgi.FieldStorage()
ctx = SimpleContext('.')
ctx.locals.form = form
for name in form.keys():
    if type(form[name]) is type([]):
        value = []
        for elem in form[name]:
            value.append(elem.value)
    else:
        value = form[name].value
    setattr(ctx.locals, name, value)
templ = ctx.load_template('form.html')
templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/form2/form.py.

You will notice that your input is sent back to you as the default value of each form element.

When you use Albatross application objects the browser request is automatically merged into the execution context for you.

5.10 More on the `<al-select>` Tag

In the previous section we performed a direct translation of the standard HTML input tags to the equivalent Albatross tags. In addition to a direct translation from the HTML form, the `<al-select>` tag supports a dynamic form.

In all but the most simple web application you will occasionally need to define the options in a `<select>` tag from internal application values. In some ways the dynamic form of the `<al-select>` tag is easier to use than the static form.

The sample program from this section is supplied in the `samples/templates/form3` directory and can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/templates/form3
python install.py
```

The form from section *Using Albatross Input Tags* has been modified to include two `<al-select>` tags, and as shown below.

```
<html>
 <head>
  <title>Display Form Input</title>
 </head>
 <body>
  Input some values to the following form and press the submit button.
  <form method="post" action="form.py">
   Text field: <al-input name="text"><br>
   Singleton checkbox: <al-input type="checkbox" name="singleton"><br>
   Checkbox group:
   <al-input type="checkbox" name="group" value="check1">
   <al-input type="checkbox" name="group" value="check2"><br>
   Radio buttons:
   <al-input type="radio" name="radio" value="radio1">
   <al-input type="radio" name="radio" value="radio2"><br>
   Option menu:
   <al-select name="select1" optionexpr="option_list1"/>
   <al-select name="select2" optionexpr="option_list2"/>
   <al-input type="submit" name="submit" value="Submit">
  </form>
  <al-include name="form-display.html">
 </body>
</html>
```

The `<al-select>` tags demonstrate the two ways that you can define option lists in your code using the `optionexpr` attribute. When converting the tag to HTML the expression in the `optionexpr` attribute is evaluated and the result is used to generate the option list that appears in the generated HTML.

The `<form.py>` program has been modified to supply lists of option values.

```
#!/usr/bin/python
import cgi
from albatross import SimpleContext

form = cgi.FieldStorage()
ctx = SimpleContext('.')

ctx.locals.option_list1 = ['option1', 'option2', 'option3']
ctx.locals.option_list2 = [(1, 'option1'), (2, 'option2'), (3, 'option3')]

ctx.locals.form = form
for name in form.keys():
    if type(form[name]) is type([]):
        value = []
        for elem in form[name]:
            value.append(elem.value)
    else:
        value = form[name].value
    setattr(ctx.locals, name, value)
templ = ctx.load_template('form.html')
templ.to_html(ctx)

print 'Content-Type: text/html'
print
ctx.flush_content()
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/form3/form.py.

If you view the browser source produced by the two `<al-select>` tags you will see the difference between the way that both option list forms are handled. Note in particular that the tuples in `option_list2` contain an integer value as the first element. This is converted to string when it is compared with the value stored in `select2` to determine which option is selected.

The browser request sent to the application will always contain strings for field values.

5.11 Streaming Application Output to the Browser

By default Albatross buffers all HTML generated from templates inside the execution context and sends a complete page once the template execution has completed (via the `flush_content()` method). The advantage of buffering the HTML is that applications can handle exceptions that occur during execution of the template and prevent any partial results leaking to the browser.

Sometimes your application needs to perform operations which take a long time. Buffering all output while lengthy processing occurs makes the application look bad. Albatross lets you use the `<al-flush>` tag to mark locations in your template file where any accumulated HTML should be flushed to the browser. The only downside to using this tag is that you lose the ability to completely insulate the user from a failure in template execution.

The sample program from this section is supplied in the `samples/templates/stream` directory and can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/templates/stream
python install.py
```

The `stream.py` program is shown below.

```
#!/usr/bin/python
import time
from albatross import SimpleContext

class SlowProcess:
    def __getitem__(self, i):
        time.sleep(1)
        if i < 10:
            return i
        raise IndexError

ctx = SimpleContext('.')
templ = ctx.load_template('stream.html')
ctx.locals.process = SlowProcess()

print 'Content-Type: text/html'
print
templ.to_html(ctx)
ctx.flush_content()
```

We have simulated a slow process by building a class that acts like a sequence of 10 elements that each take one second to retrieve.

We must make sure that we send the HTTP headers before calling the template execution (`to_html()` method). This is necessary since the execution of the template is going to cause HTML to be sent to the browser.

Now let's look at the `stream.html` template file that displays the results of our slow process.

```
<html>
 <head><title>I think I can, I think I can</title></head>
 <body>
  <p>Calculation is in progress, please stand by.
  <p>
   <al-for iter="n" expr="process">
    <al-flush>
    Stage: <al-value expr="n.value()"><br>
   </al-for>
  <p>All done!
 </body>
</html>
```

You can see the program output by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/stream/stream.py.

5.12 Displaying Tree Structured Data

One of the more powerful tags in the Albatross toolkit is the `<al-tree>` tag. This tag can be used to display almost any data that is tree structured.

The best way to explain the tag is by example. Consider the `samples/templates/tree/tree.py` sample program. This is a standalone program that is intended to be run from the command line.

```
from albatross import SimpleContext


class Node:

    def __init__(self, name, children = None):
        self.name = name
        if children is not None:
            self.children = children


ctx = SimpleContext('.')
ctx.locals.tree = Node('a', [Node('b', [Node('c'),
                                        Node('d')]),
                             Node('e', [Node('f', [Node('g', [Node('h'),
                                                               Node('i')])]),
                                        Node('j'),
                                        Node('k', [Node('l'),
                                                   Node('m')])])])
templ = ctx.load_template('tree.html')
templ.to_html(ctx)
ctx.flush_content()
```

The program constructs a tree of `Node` objects then passes the root of the tree to the `tree.html` template for formatting. The `samples/templates/tree/tree.html` template file looks like this:

```
<al-lookup name="indent">
 <al-item expr="0">  </al-item>
 <al-item expr="1"> |</al-item>
 <al-item expr="2"> \</al-item>
</al-lookup>
<al-tree iter="n" expr="tree">
 <al-for iter="c" expr="range(n.depth())">
  <al-value expr="n.line(c.value())" lookup="indent">
 </al-for>
 -<al-value expr="n.value().name" whitespace="newline">
</al-tree>
```

When you run the program it produces the following result.

```
-a
 |-b
 | |-c
 | \-d
 \-e
   |-f
   | \-g
   |   |-h
   |   \-i
   |-j
   \-k
     |-l
     \-m
```

Internally the `<al-tree>` tag uses a special iterator that is an instance of the `TreeIterator` class. This iterator performs a pre-order traversal of the tree returned by the `expr` attribute of the tag. The only requirement of the tree node objects is that child nodes are stored in the `children` sequence member.

The `<al-tree>` tag also supports a more powerful lazy loading mode of operation which is supported by Albatross application objects. Refer to section *<al-tree>*.

CHAPTER

SIX

GUIDE TO BUILDING APPLICATIONS

Roughly speaking, every page or document sent from a web server to a browser is the result of the same processing sequence. For our purposes a document is one page in an application.

1. The browser connects to the server and requests a page.
2. The server decodes the browser request and processes it. This can cause all manner of subsidiary application processing to occur.
3. The server sends the result of processing back to the browser as the next page in the application.

The essentially stateless nature of the web presents problems for the application developer who wishes to retain state between different pages in their application. From the point of view of the user, the state of the application at the server is represented by the page that they see in their browser window(s). When they enter values and press submit buttons on their page they (quite reasonably) expect the application on the web server to process their request and return the next page in the application.

The point of view of the web application developer is very different. At the server side the application must be able to receive a request from one browser, process it and send the next application page back to that browser. Before seeing the next request from the same browser, another browser may request a different page from the application. The web application even has to deal with multiple browsers simultaneously accessing different pages in the application. All of this while maintaining the illusion for the end user that they are running their own copy of the application.

Even when you only have a single machine running Apache, there are multiple Apache processes on that machine that are receiving and processing browser requests. This means that there is no guarantee that the same process that received the last request from a particular browser will receive the next request from that browser. If your application requires some state to be retained between pages, there has to be a way for that state to migrate between different Apache processes. If you are using more than one machine to process requests, then you need some way for the application state to move between machines.

There are essentially three approaches to solving the state propagation problem.

1. Deploy a stateless application. Only the most trivial applications do not require state to be retained across browser requests.
2. Get the browser to store the application state. This can be done by embedding information in URL's and hidden fields in forms. When the browser requests the next page the application restores state by extracting it from the hidden field values sent back by the browser. Browser cookies can also be used to store some types of application state.
3. Get the browser to store a session identifier in either URL's, hidden fields, or a cookie. When the browser requests the next page the application uses the browser supplied session identifier to locate the state in a server-side session database of some description.

Assuming that your application is non-trivial, the second approach (state held by the browser) is the easiest to implement.

If you are building a financial application, you will probably feel more comfortable with the third approach. This has the advantage of hiding implementation details of your application from prying eyes.

All three approaches are supported (in one way or another) by Albatross.

In all Albatross applications there are two important objects that determine how browser requests are processed; the application object, and the execution context object. The application is a potentially long lived object that provides generic services for multiple browser requests. A new execution context is created to process each browser request.

One of the things that the application and execution context do is cooperate to provide session functionality. The execution context is the object where session data is created and manipulated, but the application is usually responsible to loading and saving the session. This allows the application to maintain a long lived connection with a server if necessary.

6.1 Albatross Application Model

In the Albatross world view explained in the previous section, all web applications follow the same processing sequence to handle browser requests. When processing a browser request to generate a response the processing (in most cases) flows according to figure *Request Processing Dataflow*.

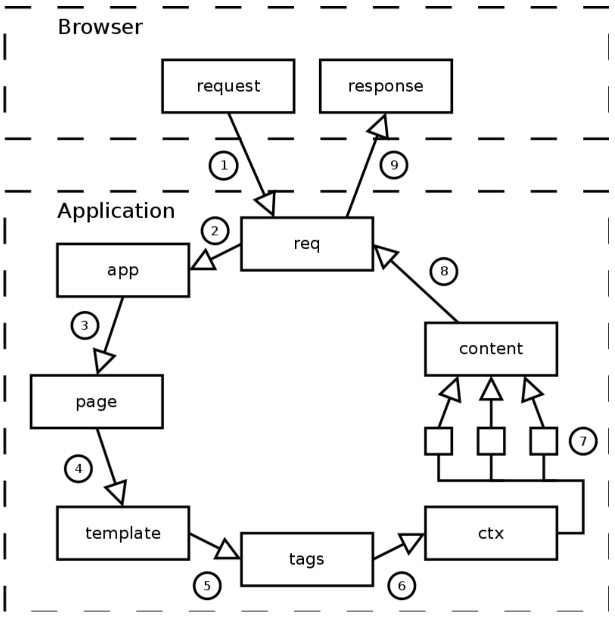


Figure 6.1: Request Processing Dataflow

The processing steps are:

1. Capture the browser request in a `Request` object.
2. Pass the `Request` object to the `run()` method of the application object.
3. Application locates the Python code for processing the browser request.
4. Page processing code runs one or more Albatross templates.
5. Templates contain either standard Albatross tags or application defined extension tags.
6. As tags are converted to HTML a stream of content fragments is sent to the execution context.
7. When the execution context content is flushed all of the fragments are joined together.
8. Joined flushed content is sent to the `Request` object `write_content()` method.
9. Application response is returned to the browser.

In the Albatross code the processing is driven by the `run()` method of the `Application` class in the `app` module.

It is instructive to look at the *exact* code from Albatross that implements the processing sequence.

```
ctx = self.create_context()
ctx.set_request(req)
self.load_session(ctx)
self.load_page(ctx)
if self.validate_request(ctx):
    self.merge_request(ctx)
    self.process_request(ctx)
self.display_response(ctx)
self.save_session(ctx)
ctx.flush_content()
```

The code is contained within a `try`/`except` block to allow the application to trap and handle exceptions.

The `Application` class assumes very little about the implementation of each of these steps. The detail of the processing stages is defined by a collection of mixin classes. A combination of the `Application` class and a selection of the mixin classes is used to construct your application class and execution context classes. There are a number of prefabricated applications and execution contexts, see chapter *Prepackaged Application and Execution Context Classes*.

This mix and match approach to building the application and execution context classes provides a great deal of flexibility. Albatross is an application toolkit, not a deployment platform. You are encouraged to look at the code and develop your own mixin classes to suit your deployment requirements. One of the primary goals of Albatross is to keep the toolkit line count low. This reduces the amount of time you need to spend before you can make your own custom extensions to the toolkit.

In the previous chapter we talked about the importance of separating the presentation layer of the application from the implementation as shown in figure *Separation of Presentation/Implementation*. Albatross HTML templates provide a fairly powerful tool for achieving that separation.

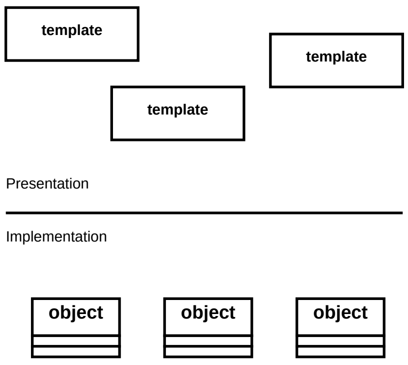


Figure 6.2: Separation of Presentation/Implementation

The presentation layer consists of a collection of template files that contain the logic required to display data from the objects contained in the implementation layer. In Albatross, the line between the two layers is the execution context.

To make objects available to the presentation layer the application places references to those objects into the local or global namespace of the execution context. The local namespace is populated in application code like this:

```
ctx.locals.mbox = Mbox(ctx.locals.username, ctx.locals.passwd)
ctx.locals.msg = ctx.locals.mbox[int(ctx.locals.msgnum) - 1]
```

To execute Python expressions inside the template files, the execution context uses the following Python code from the `NamespaceMixin` class:

```
def eval_expr(self, expr):
    self.locals.__ctx__ = self
    try:
        return eval(expr, self.__globals, self.locals.__dict__)
    finally:
        del self.locals.__ctx__
```

Whenever application code calls the `run_template()` or `run_template_once()` methods of the `NamespaceMixin` class Albatross sets the global namespace (via `set_globals()`) for expression evaluation (in `self.__globals`) to the globals of the function that called `run_template()` or `run_template_once()`.

Not many applications are output only, most accept browser input. The Albatross application object merges the browser request into the execution context in the `merge_request()` method. Referring back to the application processing sequence also note that the application object displays the result of processing the browser request via the execution context.

With this in mind figure *Separation of Presentation/Implementation* becomes figure *Presentation/Implementation and Execution Context*.

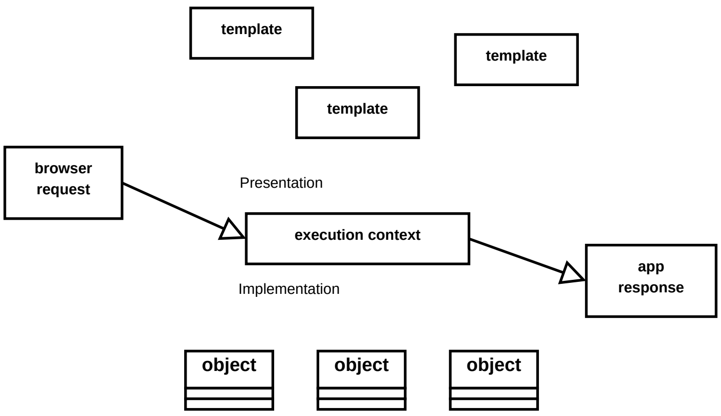


Figure 6.3: Presentation/Implementation and Execution Context

The only thing missing is the application glue that processes the browser requests, places application objects into the execution context, and directs the execution of template files.

The application model built into Albatross is intended to facilitate the use of a model-view-controller like approach (see figure *Albatross model-view-controller*) to constructing your application. There are many excellent descriptions of the model-view-controller design pattern which can be found by searching for “model view controller” on http://www.google.com/.

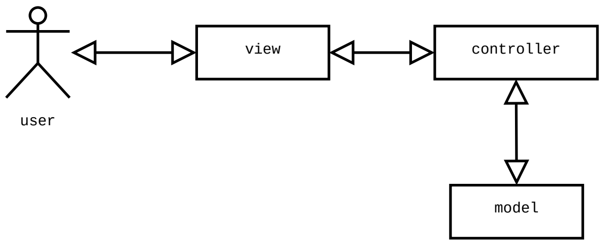


Figure 6.4: Albatross model-view-controller

The *user* invokes application functions via the *controller* through the *view*. The *controller* contains logic to direct the application functionality contained within the *model*. All of the real application functionality is in the *model*, not the *controller*. Changes to the application *model* are then propagated to the *view* via the *controller*.

In Albatross terms, the implementation layer is the *model* and the presentation layer is the *view*. The application glue plays the role of the *controller*. By divorcing all application logic from the *view* and *controller* you are able to construct unit test suites for your application functionality using the Python `unittest` module.

Albatross uses an approach inspired by the traditional model-view-controller design pattern. So now we can draw the final version of the diagram which shows how Albatross applications process browser requests in figure *Albatross Application Model*.

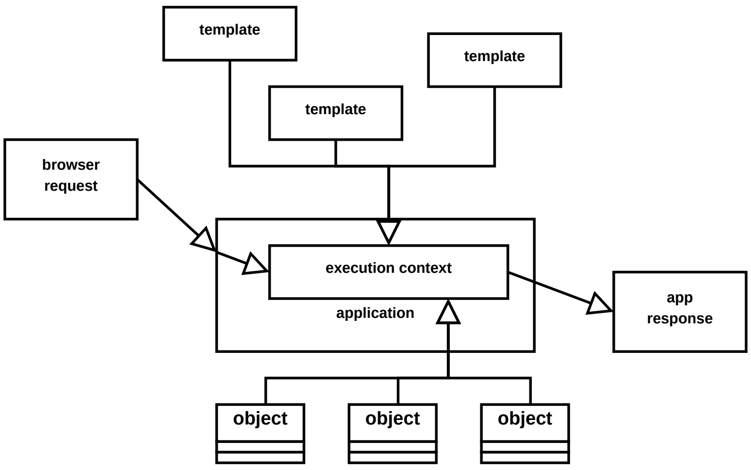


Figure 6.5: Albatross Application Model

As you can see the execution context is central to all of the Albatross processing. It is worth revisiting the application processing sequence set out in *Albatross Application Model* at the start of this section to see how the application draws all of the elements together.

During step four (page processing) the Albatross application object will call on your application code to process the browser request.

There are a number of different ways in that your application code can be "attached" to the Albatross application object. The `PageObjectMixin` application mixin class requires that you implement application functionality in "page objects" and define methods that can be called by the toolkit. As an example, here is the page object for the `'login'` page of the `popview` sample application.

```
class LoginPage:

    name = 'login'

    def page_process(self, ctx):
        if ctx.req_equals('login'):
            if ctx.locals.username and ctx.locals.passwd:
                try:
                    ctx.open_mbox()
                    ctx.add_session_vars('username', 'passwd')
                except poplib.error_proto:
                    return
                ctx.set_page('list')

    def page_display(self, ctx):
        ctx.run_template('login.html')
```

When the toolkit needs to process the browser request it calls the `page_process()` method of the current page object. As you can see, the code determines which request was made by the browser, instantiates the application objects required to service the browser request, then directs the context to move to a new page via the `set_page()` method.

When Albatross is ready to display the browser response it calls the `page_display()` method of the current page object. In the code above, `set_page()` is only called if the mailbox is opened successfully. This means that a failed login will result in the login page being displayed again.

Note that when you change pages the object that generates the HTML will be a different object to that that processed the browser request.

To let you get a foothold on the toolkit application functionality we will work through another variant of the `form` application.

6.2 Using Albatross Input Tags (Again)

In the previous chapter we demonstrated the use of Albatross input tags to transfer values from the execution context into HTML `<input>` tags, and from the browser request back into the execution context. In this section we present the same process using an Albatross application object.

The sample program from this section is supplied in the `samples/form4` directory and can be installed in your web server `cgi-bin` directory by running the following commands.

```
cd samples/form4
python install.py
```

The `form.html` template file used by the application follows.

```
<html>
 <head>
  <title>Display Form Input</title>
 </head>
 <body>
  Input some values to the following form and press the submit button.
  <al-form method="post">
   Text field: <al-input name="text"><br>
   Singleton checkbox: <al-input type="checkbox" name="singleton"><br>
   Checkbox group:
   <al-input type="checkbox" name="group" list value="check1">
   <al-input type="checkbox" name="group" list value="check2"><br>
   Radio buttons:
   <al-input type="radio" name="radio" value="radio1">
   <al-input type="radio" name="radio" value="radio2"><br>
   Option menu:
   <al-select name="select">
    <al-option>option1</al-option>
    <al-option>option2</al-option>
    <al-option>option3</al-option>
   </al-select>
   <al-input type="submit" name="submit" value="submit">
  </al-form>
  number of requests: <al-value expr="num"><br>
  text: <al-value expr="text"><br>
  singleton: <al-value expr="singleton"><br>
  group: <al-value expr="group"><br>
  radio: <al-value expr="radio"><br>
  select: <al-value expr="select"><br>
 </body>
</html>
```

The most important new features in the template file are the use of the `<al-form>` tag, and the `list` attribute in the `<al-input type="checkbox">` tag.

Most execution contexts created by application objects inherit from the `NameRecorderMixin`. The `NameRecorderMixin` records the name, type and multiple value disposition of each input tag in a form in a cryptographically signed hidden field named `__albform__`. This mechanism prevents clients from being able to merge arbitrary data into the local namespace, as well as providing additional information to make the merging process more reliable. The recording process requires that all `<al-input>` tags be enclosed by an `<al-form>` tag.

When the resulting form is submitted, the contents of the `__albform__` field controls merging of form fields into `ctx.locals`. Only fields tracked by `__albform__` will be merged. Consequently, if submission occurs via a GET request without an `__albform__` field (for example, as a result of the user following an `<al-a>`), the application must explicitly request relevent fields be merged via the `merge_vars(...)` method. [1]

Any input field with the `list` attribute will always receive a list value from a POST browser request regardless of how many values (including none) were sent by the browser. An exception will be raised if you specify multiple input tags with the same name in a form and do not include the `list` attribute. The input tag types `radio`, `image`, and `submit` can only have a single value, even if multiple inputs of the same name appear in a form, and the `list` attribute should not be specified on these.

The `form.py` program is show below.

```
#!/usr/bin/python
from albatross import SimpleApp
from albatross.cgiapp import Request


class Form:

    def page_enter(self, ctx):
        ctx.locals.text = ctx.locals.singleton = ctx.locals.group = \
                          ctx.locals.radio = ctx.locals.select = None
        ctx.locals.num = 0
        ctx.add_session_vars('num')

    def page_display(self, ctx):
        ctx.locals.num += 1
        ctx.run_template('form.html')


app = SimpleApp(base_url='form.py',
                template_path='.',
                start_page='form',
                secret='-=-secret-=-')
app.register_page('form', Form())


if __name__ == '__main__':
    app.run(Request())
```

You can run the program by pointing your browser at http://www.object-craft.com.au/cgi-bin/alsamp/form4/form.py.

Notice that the browser request is automatically merged into the local namespace and then extracted by the template when generating the HTML response.

The program uses the `SimpleApp` application class. `SimpleApp` uses an object to define each page served by the application. Each of the page objects must be registered with the application via the `register_page()` method.

When the application enters a new page `SimpleApp` calls the `page_enter()` method of the page object to allow the application to initialise execution context values. In the above program the `page_enter()` method initialises all values used by the HTML form to `None`, initialises the variable *num* to `0` and places it into the session.

As shown in the application processing sequence in the *Albatross Application Model* section, the first step in handling a browser request is to create an execution context. The `SimpleApp` class uses instances of the `SimpleAppContext` class which inherits from `HiddenFieldSessionMixin`. The `HiddenFieldSessionMixin` class stores session data in a hidden field named `__albstate__` at the end of each form.

When an Albatross application needs to display the result of processing a request it calls the `page_display()` method of the current page. In the above program this method increments *num* and then runs the `form.html`

[1] Prior to Albatross version 1.36, in the absence of an `__albform__` field, all request fields would be merged.

template. It is important to note that any changes to session values after executing a template will be lost as the session state is saved in the HTML produced by the template.

It is worth explaining again that the program does not perform any request merging — this is all done automatically by the Albatross application and execution context objects.

6.3 The Popview Application

In this section we will develop a simple application that allows a user to log onto a POP server to view the contents of their mailbox. Python provides the `poplib` module which provides a nice interface to the POP3 protocol.

The complete sample program is contained in the `samples/popview1` directory. Use the `install.py` script to install the sample.

```
cd samples/popview1
python install.py
```

First of all let's create the *model* components of the application. This consists of some classes to simplify access to a user mailbox. Create a module called `popviewlib.py` and start with the `Mbox` class.

```
import string
import poplib

pophost = 'pop'


class Mbox:

    def __init__(self, name, passwd):
        self.mbox = poplib.POP3(pophost)
        self.mbox.user(name)
        self.mbox.pass_(passwd)

    def __getitem__(self, i):
        try:
            return Msg(self.mbox, i + 1)
        except poplib.error_proto:
            raise IndexError
```

The important feature of our `Mbox` class is that it implements the Python sequence protocol to retrieve messages. This allows us to iterate over the mailbox using the `<al-for>` tag in templates. When there is an attempt to retrieve a message number which does not exist in the mailbox, a `poplib.error_proto` exception will be raised. We transform this exception into an `IndexError` exception to signal the end of the sequence.

The sequence protocol is just one of the truly excellent Python features which allows your application objects to become first class citizens.

Next we need to implement a `Msg` class to access the header and body of each message.

```
class Msg:

    def __init__(self, mbox, msgnum):
        self.mbox = mbox
        self.msgnum = msgnum
        self.read_headers()

    def read_headers(self):
        res = self.mbox.top(self.msgnum, 0)
        hdrs = Headers()
        hdr = None
        for line in res[1]:
            if line and line[0] in string.whitespace:
                hdr = hdr + '\n' + line
            else:
                hdrs.append(hdr)
                hdr = line
        hdrs.append(hdr)
        self.hdrs = hdrs
        return hdrs

    def read_body(self):
        res = self.mbox.retr(self.msgnum)
        lines = res[1]
        for i in range(len(lines)):
            if not lines[i]:
                break
        self.body = string.join(lines[i:], '\n')
        return self.body
```

Note that we retrieve the message headers in the constructor to ensure that the creation of the `Msg` object will fail if there is no such message in the mailbox. The `Mbox` class uses the exception raised by the `poplib` module to detect when a non-existent message is referenced.

The `poplib` module returns the message headers as a flat list of text lines. It is up to the `poplib` user to process those lines and impose a higher level structure upon them. The `read_headers()` method attaches header continuation lines to the corresponding header line before passing each complete header to a `Headers` object.

The `read_body()` method retrieves the message body lines from the POP server and combines them into a single string.

We are going to need to display message headers by name so our `Headers` class implements the dictionary protocol.

```
class Headers:

    def __init__(self):
        self.hdrs = {}

    def append(self, header):
        if not header:
            return
        parts = string.split(header, ': ', 1)
        name = string.capitalize(parts[0])
        if len(parts) > 1:
            value = parts[1]
        else:
            value = ''
        curr = self.hdrs.get(name)
        if not curr:
            self.hdrs[name] = value
            return
        if type(curr) is type(''):
            curr = self.hdrs[name] = [curr]
        curr.append(value)

    def __getitem__(self, name):
        return self.hdrs.get(string.capitalize(name), '')
```

Instead of raising a `KeyError` for undefined headers, the `__getitem__()` method returns the empty string. This allows us to test the presence of headers in template files without being exposed to exception handling.

Lets take all of these classes for a spin in the Python interpreter.

```
>>> import popviewlib
>>> mbox = popviewlib.Mbox('djc', '***')
>>> msg = mbox[0]
>>> msg.hdrs['From']
'Owen Taylor <otaylor@redhat.com>'
>>> print msg.read_body()

Daniel Egger <egger@suse.de> writes:

> Am 05 Aug 2001 12:00:15 -0400 schrieb Alex Larsson:
>
[snip]
```

Next we will create the application components (the *controller*) in `popview.py`. Albatross has a prepackaged application object which you can use for small applications; the `SimpleApp` class.

In anything but the most trivial applications it is probably a good idea to draw a site map like figure *Popview Site Map* to help visualise the application.

Our application will contain three pages; login, message list, and message detail.

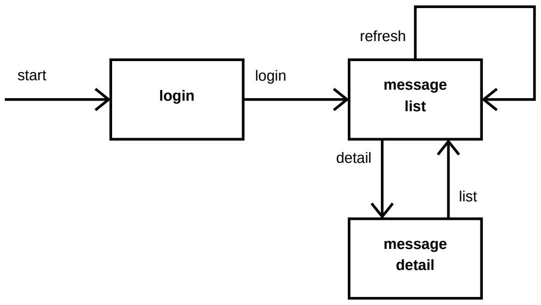


Figure 6.6: Popview Site Map

The `SimpleApp` class inherits from the `PageObjectMixin` class which requires that the application define an object for each page in the application.

Albatross stores the current application page identifier in the local namespace of the execution context as `__page__`. The value is automatically created and placed into the session.

When the current page changes, the Albatross application object calls the `page_enter()` function/method of the new page object. The `page_process()` method is called to process a browser request, and finally `page_display()` is called to generate the application response.

The `SimpleApp` class creates `SimpleAppContext` execution context objects. By subclassing `SimpleApp` and overriding `create_context()` we can define our own execution context class.

The prologue of the application imports the required application and execution context classes from Albatross and the `Request` class from the deployment module.

```python
#!/usr/bin/python
from albatross import SimpleApp, SimpleAppContext
from albatross.cgiapp import Request
import poplib
import popviewlib
```

Next in the file is the class for the login page. Note that we only implement glue (or *controller*) logic in the page objects. Each time a new page is served we will need to open the mail box and retrieve the relevant data. That means that the username and password will need to be stored in some type of session data storage.

```
class LoginPage:

    name = 'login'

    def page_process(self, ctx):
        if ctx.req_equals('login'):
            if ctx.locals.username and ctx.locals.passwd:
                try:
                    ctx.open_mbox()
                    ctx.add_session_vars('username', 'passwd')
                except poplib.error_proto:
                    return
                ctx.set_page('list')

    def page_display(self, ctx):
        ctx.run_template('login.html')
```

The `req_equals()` method of the execution context looks inside the browser request for a field with the specified name. It returns a TRUE value if such a field exists and it has a value not equal to `None`. The test above will detect when a user presses the submit button named `'login'` on the `login.html` page.

Next, here is the page object for displaying the list of messages in the mailbox.

```
class ListPage:

    name = 'list'

    def page_process(self, ctx):
        if ctx.req_equals('detail'):
            ctx.set_page('detail')

    def page_display(self, ctx):
        ctx.open_mbox()
        ctx.run_template('list.html')
```

The “detail” page displays the message detail.

```
class DetailPage:

    name = 'detail'

    def page_process(self, ctx):
        if ctx.req_equals('list'):
            ctx.set_page('list')

    def page_display(self, ctx):
        ctx.open_mbox()
        ctx.read_msg()
        ctx.run_template('detail.html')
```

And finally we define the application class and instantiate the application object. Note that we have subclassed `SimpleApp` to create our own application class. This allows us to implement our own application level functionality as required.

```
class AppContext(SimpleAppContext):

    def open_mbox(self):
        if hasattr(self.locals, 'mbox'):
            return
        self.locals.mbox = popviewlib.Mbox(self.locals.username, self.locals.passwd)

    def read_msg(self):
        if hasattr(self.locals, 'msg'):
            return
        self.locals.msg = self.locals.mbox[int(self.locals.msgnum) - 1]
        self.locals.msg.read_body()


class App(SimpleApp):

    def __init__(self):
        SimpleApp.__init__(self,
                           base_url='popview.py',
                           template_path='.',
                           start_page='login',
                           secret='-=-secret-=-')
        for page_class in (LoginPage, ListPage, DetailPage):
            self.register_page(page_class.name, page_class())

    def create_context(self):
        return AppContext(self)


app = App()


if __name__ == '__main__':
    app.run(Request())
```

The *base_url* argument to the application object constructor will be placed into the `action` attribute of all forms produced by the `<al-form>` tag. It will also form the left hand side of all hrefs produced by the `<al-a>` tag. The *template_path* argument is a relative path to the directory that contains the application template files. The *start_page* argument is the name of the application start page. When a browser starts a new session with the application it will be served the application start page.

We have also created our own execution context to provide some application functionality as execution context methods.

With the *model* and *controller* components in place we can now move onto the template files that comprise the *view* components of the application.

First let's look at the `login.html` page.

```
<html>
 <head>
  <title>Please log in</title>
 </head>
 <body>
  <h1>Please log in</h1>
  <al-form method="post">
   <table>
    <tr>
     <td>Username</td>
     <td><al-input name="username" size="10" maxlength="10"></td>
    </tr>
    <tr>
     <td>Password</td>
     <td><al-input type="password" name="passwd" size="10" maxlength="10"></td>
     <td><al-input type="submit" name="login" value="Log In"></td>
    </tr>
   </table>
  </al-form>
 </body>
</html>
```

When you look at the HTML produced by the application you will notice two extra `<input>` tags have been generated at the bottom of the form. They are displayed below (reformatted to fit on the page).

```
<input type="hidden" name="__albform__" value="eJzTDJeu3P90rZC6dde04xUhHL
WFjBqhHKXFqUV5ibmphUzeDKFsBYnFxeUphcxANmtOfnpmXiGLN0OpHgB7UBOp
">
<input type="hidden" name="__albstate__" value="eJzT2sr5Jezh942TUrMty6q1j
WsLGUM54uMLEtNT4+MLmUJZc/LTM/MKmYv1AH8XEAY=
">
```

If we fire up the Python interpreter we can have a look at what these fields contain.

```
>>> import base64,zlib,cPickle
>>> s = "eJzTDJeu3P90rZC6dde04xUhHL\n" + \
... "WFjBqhHKXFqUV5ibmphUzeDKFsBYnFxeUphcxANmtOfnpmXiGLN0OpHgB7UBOp\n"
>>> cPickle.loads(zlib.decompress(base64.decodestring(s))[16:])
{'username': 0, 'passwd': 0, 'login': 0}
>>> s = "eJzT2sr5Jezh942TUrMty6q1j\n" + \
... "WsLGUM54uMLEtNT4+MLmUJZc/LTM/MKmYv1AH8XEAY=\n"
>>> cPickle.loads(zlib.decompress(base64.decodestring(s))[16:])
{'__page__': 'login'}
```

The first string contains a dictionary that defines the name and type of the input fields that were present in the form. This is placed into the form by the `NameRecorderMixin` class which is subclassed by `SimpleAppContext`. If you look at the definition of `SimpleAppContext` you will notice the following definition at the start of the class.

```
NORMAL = 0
MULTI = 1
MULTISINGLE = 2
FILE = 3
```

The value `0` for each field in the dictionary above corresponds to field type `NORMAL`.

When merging the browser request into the execution context the dictionary of field names is used to assign the value `None` to any `NORMAL` or `FILE` fields, or `[]` to any `LIST` fields that were left empty by the user. This is useful because it lets us write application code which can ignore the fact that fields left empty by the user will not be sent by the browser when the form is submitted.

The second string contains all of the session values for the application. In the application start page the only session variable that exists is the __page__ variable. The `HiddenFieldSessionMixin` places this field in the form when the template is executed and pulls the field value back out of the browser request into the execution context when the session is loaded.

The first 20 bytes of the decompressed string is an HMAC-SHA1 cryptographic signature. This was generated by combining the application secret (passed as the *secret* argument to the application object) with the pickle string with a cryptographic hash. When the field is sent back to the application the signing process is repeated. The pickle is only loaded if the sign sent by the browser and the regenerated signature are the same.

Since the `popview` application has been provided for example purposes you will probably forgive the usage of the `HiddenFieldSessionMixin` class to propagate session state. In a real application that placed usernames and passwords in the session you would probably do something to protect these values from prying eyes.

Now let's look at the `list.html` template file.

Note that the use of the `<al-flush>` tag causes the HTML output to be streamed to the browser. Use of this tag can give your application a much more responsive feel when generating pages that involve lengthy processing.

A slightly obscure feature of the page is the use of a separate form surrounding the `<al-input type="image">` field used to select each message. An unfortunate limitation of the HTML `<input type="image">` tag is that you cannot associate a value with the field because the browser returns the coordinates where the user pressed the mouse inside the image. In order to associate the message number with the image button we place the message number in a separate hidden `<input>` field and group the two fields using a form.

You have to be careful creating a large number of forms on the page because each of these forms will also contain the `__albform__` and `__albstate__` hidden fields. If you have a lot of data in your session the size of the `__albstate__` field will cause the size of the generated HTML to explode.

```
<html>
 <head>
  <title>Message List</title>
 </head>
 <body>
  <al-form method="post">
   <al-input type="submit" name="refresh" value="Refresh">
  </al-form>
  <hr noshade>
  <table>
   <tr align="left">
    <td><b>View</b></td>
    <td><b>To</b></td>
    <td><b>From</b></td>
    <td><b>Subject</b></td>
   </tr>
   <al-for iter="m" expr="mbox">
   <tr align="left" valign="top">
    <td>
     <al-form method="post">
      <al-input type="image" name="detail" src="/icons/generic.gif" border="0">
      <al-input type="hidden" name="msgnum" expr="m.value().msgnum">
     </al-form>
    </td>
    <td><al-value expr="m.value().hdrs['To']"></td>
    <td><al-value expr="m.value().hdrs['From']"></td>
    <td><al-value expr="m.value().hdrs['Subject']"></td>
   </tr>
   <al-if expr="(m.index() % 10) == 9"><al-flush></al-if>
   </al-for>
  </table>
 </body>
</html>
```

Finally, here is the message detail page `detail.html`.

```
<html>
 <head>
  <title>Message Detail</title>
 </head>
 <body>
  <al-form method="post">
   <al-input type="submit" name="list" value="Back">
  </al-form>
  <hr noshade>
  <table>
   <al-for iter="f" expr="('To', 'Cc', 'From', 'Subject')">
    <al-if expr="msg.hdrs[f.value()]">
     <tr align="left">
      <td><b><al-value expr="f.value()">:</b></td>
      <td><al-value expr="msg.hdrs[f.value()]"></td>
     </tr>
    </al-if>
   </al-for>
  </table>
  <hr noshade>
```

```
    <pre><al-value expr="msg.body"></pre>
   </body>
  </html>
```

6.4 Adding Pagination Support to Popview

If the previous section we constructed a simple application for viewing the contents of a mailbox via the `poplib` module. One problem with the application is that there is no limit to the number of messages that will be displayed. In this section we will build pagination support into the message list page.

We will modify the application to display 15 messages on each page and will add buttons to navigate to the next and previous pages. The `pagesize` attribute of the `<al-for>` tag provides automatic pagination support for displaying sequences. The only extra code that we need to add is a `__len__` method to the `Mbox` class which returns the number of messages in the mailbox. This `__len__` method is needed to allow the template file to test whether or not to display a next page control.

The complete sample program is contained in the `samples/popview2` directory. Use the `install.py` script to install the sample.

```
cd samples/popview2
python install.py
```

Add a `__len__` method to the `Mbox` class in `popviewlib.py`.

```
class Mbox:

    def __init__(self, name, passwd):
        self.mbox = poplib.POP3(pophost)
        self.mbox.user(name)
        self.mbox.pass_(passwd)

    def __getitem__(self, i):
        try:
            return Msg(self.mbox, i + 1)
        except poplib.error_proto:
            raise IndexError

    def __len__(self):
        len, size = self.mbox.stat()
        return len
```

Now we modify the template file `list.html`.

```
<html>
 <head>
  <title>Message List</title>
 </head>
 <body>
  <al-for iter="m" expr="mbox" pagesize="15" prepare/>
  <al-form method="post">
   <al-if expr="m.has_prevpage()">
    <al-input type="image" prevpage="m" src="/icons/left.gif" border="0">
   </al-if>
   <al-input type="submit" name="refresh" value="Refresh">
   <al-if expr="m.has_nextpage()">
    <al-input type="image" nextpage="m" src="/icons/right.gif" border="0">
   </al-if>
  </al-form>
  <hr noshade>
```

```
      <table>
       <tr align="left">
        <td></td>
        <td><b>View</b></td>
        <td><b>To</b></td>
        <td><b>From</b></td>
        <td><b>Subject</b></td>
       </tr>
       <al-for iter="m" continue>
       <tr align="left" valign="top">
        <td><al-value expr="m.value().msgnum"></td>
        <td>
         <al-form method="post">
          <al-input type="image" name="detail" src="/icons/generic.gif" border="0">
          <al-input type="hidden" name="msgnum" expr="m.value().msgnum">
         </al-form>
        </td>
        <td><al-value expr="m.value().hdrs['To']"></td>
        <td><al-value expr="m.value().hdrs['From']"></td>
        <td><al-value expr="m.value().hdrs['Subject']"></td>
       </tr>
       </al-for>
      </table>
      <hr noshade>
     </body>
    </html>
```

The `<al-for>` tag just below the `<body>` tag contains two new attributes; `pagesize` and `prepare`.

The `pagesize` turns on pagination for the `<al-for>` `ListIterator` object and defines the size of each page. In order to remember the current top of page between pages, the tag places the iterator into the session. When saving the iterator, only the top of page and pagesize are retained.

The `prepare` attribute instructs the `<al-for>` tag to perform all tasks except actually display the content of the sequence. This allows us to place pagination controls before the actual display of the list.

6.5 Adding Server-Side Session Support to Popview

So far we have been saving all application state at the browser inside hidden fields. Sometimes it is preferable to retain state at the server side. Albatross includes support for server-side sessions in the `SessionServerContextMixin` and `SessionServerAppMixin` classes. In this section we will modify the `popview.py` program to use server-side sessions.

The `SessionServerAppMixin` class uses a socket to communicate with the `session-server/al-session-daemon` session server. You will need to start this program before using the new `popview` application.

In previous versions of the program we were careful to place all user response into forms. This allowed Albatross to transparently attach the session state to hidden fields inside the form. When using server-side sessions Albatross does not need to save any application state at the browser so we are free to use URL style user inputs. To illustrate the point, we will replace all of the form inputs with URL user inputs.

The complete sample program is contained in the `samples/popview3` directory. Use the `install.py` script to install the sample.

```
cd samples/popview3
python install.py
```

The new `list.html` template file follows.

```
<html>
 <head>
  <title>Message List</title>
 </head>
 <body>
  <al-for iter="m" expr="mbox" pagesize="15" prepare/>
  <al-if expr="m.has_prevpage()"><al-a prevpage="m">Prev</al-a> | </al-if>
  <al-a href="refresh=1">Refresh</al-a>
  <al-if expr="m.has_nextpage()"> | <al-a nextpage="m">Next</al-a></al-if>
  <hr noshade>
  <table>
   <tr align="left">
    <td></td>
    <td><b>To</b></td>
    <td><b>From</b></td>
    <td><b>Subject</b></td>
   </tr>
   <al-for iter="m" continue>
   <tr align="left" valign="top">
    <td align="right">
     <al-a expr="'msgnum=%s' % m.value().msgnum">
      <al-value expr="m.value().msgnum"></al-a>
    </td>
    <td><al-value expr="m.value().hdrs['To']"></td>
    <td><al-value expr="m.value().hdrs['From']"></td>
    <td><al-value expr="m.value().hdrs['Subject']"></td>
   </tr>
   </al-for>
  </table>
  <hr noshade>
 </body>
</html>
```

Next the new `detail.html` template file.

```
<html>
 <head>
  <title>Message Detail</title>
 </head>
 <body>
  <al-a href="list=1">Back</al-a>
  <hr noshade>
  <table>
   <al-for iter="f" expr="('To', 'Cc', 'From', 'Subject')">
    <al-if expr="msg.hdrs[f.value()]">
     <tr align="left">
      <td><b><al-value expr="f.value()">:</b></td>
      <td><al-value expr="msg.hdrs[f.value()]"></td>
     </tr>
    </al-if>
   </al-for>
  </table>
  <hr noshade>
  <pre><al-value expr="msg.body"></pre>
 </body>
</html>
```

One of the more difficult tasks for developing stateful web applications is dealing with browser requests submitted from old pages in the browser history.

When all application state is stored in hidden fields in the HTML, requests from old pages do not usually cause problems. This is because the old application state is provided in the same request.

When application state is maintained at the server, requests from old pages can cause all sorts of problems. The current application state at the server represents the result of a sequence of browser requests. If the user submits a request from an old page in the browser history then the fields and values in the request will probably not be relevant to the current application state

Making sure all application requests are uniquely named provides some protection against the application processing a request from another page which just happened the share the same request name. It is not a complete defense as you may receive a request from an old version of the same page. A request from an old version of a page is likely to make reference to values which no longer exist in the server session.

Some online banking applications attempt to avoid this problem by opening browser windows that do not have history navigation controls. A user who uses keyboard accelerators for history navigation will not be hindered by the lack of navigation buttons.

The `popview` application does not modify any of the data it uses so there is little scope for submissions from old pages to cause errors.

By changing the base class for the application object we can gain support for server side sessions. Albatross includes a simple session server and supporting mixin classes.

The new application prologue looks like this:

```
#!/usr/bin/python
from albatross import SimpleSessionApp, SessionAppContext
from albatross.cgiapp import Request
import popviewlib
```

The execution context now inherits from `SessionAppContext`:

```
class AppContext(SessionAppContext):

    def open_mbox(self):
        if hasattr(self.locals, 'mbox'):
            return
        self.locals.mbox = popviewlib.Mbox(self.locals.username, self.locals.passwd)

    def read_msg(self):
        if hasattr(self.locals, 'msg'):
            return
        self.locals.msg = self.locals.mbox[int(self.locals.msgnum) - 1]
        self.locals.msg.read_body()
```

And the new application class looks like this:

```
class App(SimpleSessionApp):

    def __init__(self):
        SimpleSessionApp.__init__(self,
                                  base_url='popview.py',
                                  template_path='.',
                                  start_page='login',
                                  secret='-=-secret-=-',
                                  session_appid='popview3')
        for page_class in (LoginPage, ListPage, DetailPage):
            self.register_page(page_class.name, page_class())

    def create_context(self):
        return AppContext(self)
```

The *session_appid* argument to the constructor is used to uniquely identify the application at the server so that multiple applications can be accessed from the same browser without the session from one application modifying the session from another.

Apart from changes to load the new template files, we also need to change the ListPage class because we changed the method of selecting messages from the message list.

```
class ListPage:

    name = 'list'

    def page_process(self, ctx):
        if ctx.req_equals('msgnum'):
            ctx.set_page('detail')

    def page_display(self, ctx):
        ctx.open_mbox()
        ctx.run_template('list.html')
```

6.6 Building Applications with Page Modules

Implementing an application as a monolithic program is fine for small applications. As the application grows the startup time becomes an issue, as does maintenance. Albatross provides a set of classes that allow you to implement each page in a separate Python module. In this section we will convert the popview application to this type of application.

Converting a monolithic application to a page module application is usually fairly simple. First we must turn each page object into a page module. When we used page objects, the class that implemented each page was identified by the name class member. With page modules the name of the module identifies the page that it processes.

The complete sample program is contained in the samples/popview4 directory. Use the install.py script to install the sample.

```
cd samples/popview4
python install.py
```

The LoginPage class becomes login.py.

```
import poplib


def page_process(ctx):
    if ctx.req_equals('login'):
        if ctx.locals.username and ctx.locals.passwd:
            try:
                ctx.open_mbox()
                ctx.add_session_vars('username', 'passwd')
            except poplib.error_proto:
                return
            ctx.set_page('list')


def page_display(ctx):
    ctx.run_template('login.html')
```

The ListPage class becomes list.py.

```
def page_process(ctx):
    if ctx.req_equals('msgnum'):
        ctx.set_page('detail')


def page_display(ctx):
    ctx.open_mbox()
    ctx.run_template('list.html')
```

And the `DetailPage` class becomes `detail.py`.

```
def page_process(ctx):
    if ctx.req_equals('list'):
        ctx.set_page('list')


def page_display(ctx):
    ctx.open_mbox()
    ctx.read_msg()
    ctx.run_template('detail.html')
```

When using page modules we do not need to register each page module. When Albatross needs to locate the code for a page it simply imports the module. So the entire `popview.py` program now looks like this:

```
#!/usr/bin/python
from albatross import ModularSessionApp, SessionAppContext
from albatross.cgiapp import Request
import popviewlib


class AppContext(SessionAppContext):

    def open_mbox(self):
        if hasattr(self.locals, 'mbox'):
            return
        self.locals.mbox = popviewlib.Mbox(self.locals.username, self.locals.passwd)

    def read_msg(self):
        if hasattr(self.locals, 'msg'):
            return
        self.locals.msg = self.locals.mbox[int(self.locals.msgnum) - 1]
        self.locals.msg.read_body()


class App(ModularSessionApp):

    def __init__(self):
        ModularSessionApp.__init__(self,
                                   base_url='popview.py',
                                   module_path='.',
                                   template_path='.',
                                   start_page='login',
                                   secret='-=-secret-=-',
                                   session_appid='popview4')

    def create_context(self):
        return AppContext(self)


app = App()


if __name__ == '__main__':
    app.run(Request())
```

6.7 Random Access Applications

In the popview application the server is in complete control of the sequence of pages that are served to the browser. In some applications you want the user to be able to bookmark individual pages for later retrieval in any desired sequence. Albatross provides application classes built with the `RandomPageModuleMixin` class for this very purpose.

The `random` sample is provided to demonstrate the use of the `RandomModularSessionApp` class. Use the `install.py` script to install the sample.

```
cd samples/random
python install.py
```

The complete mainline of the `randompage.py` sample is shown below.

```
#!/usr/bin/python
from albatross import RandomModularSessionApp
from albatross.cgiapp import Request


app = RandomModularSessionApp(base_url='randompage.py',
                              page_path='pages',
                              start_page='tree',
                              secret='-=-secret-=-',
                              session_appid='random')


if __name__ == '__main__':
    app.run(Request())
```

When processing the browser request the application determines which page to serve to the browser by inspecting the URL in the browser request. The page identifier is taken from the part of the URL which follows the *base_url* argument to the constructor. If the page identifier is empty then the application serves the page identified by the *start_page* argument to the constructor.

If you point your browser at http://www.object-craft.com.au/cgi-bin/alsamp/random/randompage.py you will notice that the server has redirected your browser to the start page.

The sample program defines two pages which demonstrate two different ways to direct user navigation through the application.

The `tree.html` page template uses a form to capture user input.

```
<html>
<head><title>Here is a Tree</title></head>
<body>
<al-lookup name="indent">
 <al-item expr="0">  </al-item>
 <al-item expr="1"> |</al-item>
 <al-item expr="2"> \</al-item>
</al-lookup>
<h1>Here is a Tree</h1>
<al-form method="post">
<pre>
<al-tree iter="n" expr="tree">
 <al-for iter="c" expr="range(n.depth())">
  <al-value expr="n.line(c.value())" lookup="indent">
 </al-for>
 -<al-input type="checkbox" alias="n.value().selected">
  <al-value expr="n.value().name" whitespace="newline">
</al-tree>
</pre>
```

```
<al-input type="submit" name="save" value="Save">
<al-input type="submit" name="paginate" value="To Paginate Page">
</al-form>
</body>
</html>
```

During conversion to HTML the `<al-form>` tag automatically places the name of the current page into the `action` attribute. This makes the browser send the response back to the same page module (`tree.py`).

```
from utils import Node

def page_display(ctx):
    ctx.run_template('tree.html')


def page_process(ctx):
    if ctx.req_equals('paginate'):
        ctx.redirect('paginate')
    if not hasattr(ctx.locals, 'tree'):
        ctx.locals.tree \
            = Node('a', [Node('a', [Node('a'),
                                    Node('b')]),
                         Node('b', [Node('a', [Node('a', [Node('a'),
                                                          Node('b')])]),
                                    Node('b'),
                                    Node('c', [Node('a'),
                                               Node('b')])])])
        ctx.add_session_vars('tree')
```

When the application receives the `paginate` request it uses the `redirect()` method to direct the browser to a new page.

The `paginate.html` page template uses a URLs to capture user input.

```
<html>
 <head><title>Pagination Example</title></head>
 <body>
  <h1>This is a list with pagination...</h1>

  <al-for iter="i" expr="data" pagesize="10" prepare/>
  <al-if expr="i.has_prevpage()">
   <al-a prevpage="i">prev</al-a>
  </al-if>
  <al-for iter="i" pagesize="10">
   <al-if expr="i.count()">,</al-if> <al-value expr="i.value()">
  </al-for>
  <al-if expr="i.has_nextpage()" whitespace>
   <al-a nextpage="i">next</al-a>
  </al-if>
  <p>
  <al-a href="tree">To Tree Page</al-a>
 </body>
</html>
```

During conversion to HTML the `<al-a>` tag automatically translates the `href="tree"` attribute into a URL which requests the `tree` page from the application. Since the browser is doing all of the work, the `paginate.py` module which handles the `paginate` page is very simple.

```
def page_display(ctx):
    ctx.locals.data = range(100)
    ctx.run_template('paginate.html')
```

6.8 The Albatross Session Server

The Albatross Session server works in concert with the execution context and application mixin classes in the `albatross.session` module to provide server-side session recording. The application and the server communicate via TCP sockets. By default, session servers listen on port 34343.

More than one Albatross application can share a single session server process, and applications can be deployed over multiple web server hosts.

6.8.1 Sample Simple Session Server

The `albatross.simpleserver` module is a sample session server. It can either be used stand-alone, or imported into other Python scripts.

The session server uses TCP sockets to communicate with the application mixin, by default listening on port 34343. The server port can be changed by using the `-p` or `--port=` command line argument. Internally the server uses a select loop to allow connections from multiple applications simultaneously.

Note that the daemon does not need to run as `root`, provided it listens on a port above 1024. If possible, you should run it under a user ID not shared by any other processes (and not `nobody`). You should also ensure that only authorised clients can connect to your session server, as the protocol provides no authentication or authorisation mechanisms.

Application constructor arguments which are relevant to the session server are:

- *session_appid*

  This is used to identify the application with the session server. It is also used as the session id in the cookie sent to the browser.

- *session_server* = `'localhost'`

  If you decide to run the session server on a different machine to the application you must pass the host name of the session server in this argument.

- *server_port* = `34343`

  If you decide to run the session server on a different port you must pass the port number in this argument.

- *session_age* = `1800`

  This argument defines the amount of time in seconds for which idle sessions will kept in the server.

6.8.2 Unix Session Server Daemon

In the `session-server` directory of the source distribution is a Unix daemon version of the `simpleserver.py` session server, called `al-session-daemon`.

Long running server processes under Unix need to be backgrounded, and disassociated from the terminal device on which they were started. `al-session-daemon` provides these functions, as well as recording the process ID of the daemon so as to allow it to be easily shut-down.

Note that the daemon does not need to run as `root`, provided it listens on a port above 1024, and can write to it's pid and log directories. If possible, you should run it under a user ID not shared by any other processes (and not `nobody`). You should also ensure that only authorised clients can connect to your session server, as the protocol provides no authentication or authorisation mechanisms.

```
Usage: al-session-daemon [options]... <command>

Where [options] are:
  -D, --debug                     Write debugging to log
  -h, --help                      Display this help and exit
  -k <pid-file>,                  Record server pid in <pid-file>, default is
```

```
    --pidfile=<pid-file>        /var/run/al-session-daemon.pid
  -p <port>, --port=<port>      Listen on <port>, default is 34343
  -l <log-file>,                Write log to <log-file>, default is
    --log=<log-file>            /var/log/al-session-daemon.log

<command> is one of:
  start    start a new daemon
  stop     kill the current daemon
  status   request the daemon's status
```

6.8.3 Server Protocol

You can see the session server in action by using telnet.

```
djc@rat:~$ telnet localhost 34343
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
new myapp 3600
OK 38b80b3f546c8cfa
put myapp 38b80b3f546c8cfa
OK - send data now, terminate with blank line
here is my session data
it is on multiple lines

OK
get myapp 38b80b3f546c8cfa
OK - session follows
here is my session data
it is on multiple lines

del myapp 38b80b3f546c8cfa
OK
get myapp 38b80b3f546c8cfa
ERROR no such session
quit
Connection closed by foreign host.
djc@rat:~$
```

All dialogue is line based using a CRLF end of line sequence. Session ids are generated by the server and each application has its own set of session ids. The application mixin class in the `albatross.session` module uses the *session_appid* argument to the constructor as the application id with the session server. Note that this application id is also used in the cookie sent to the browser.

If a command was successful the server response line will start with the text `'OK'` otherwise it will start with `'ERROR'`.

Create New Session

To create a new session for the *appid* application which will be deleted if left idle for more than *age* seconds the application sends a line of the form:

```
"new " appid " " age CRLF
```

Successful response will be a line of the form:

```
"OK " sesid CRLF
```

Save Session

To save data into an existing session the application sends a line of the form:

```
"put " appid " " sesid CRLF
```

If the session exists in the server it will respond with the following line:

```
"OK - send data now, terminate with blank line" CRLF
```

The program then sends a sequence of text lines terminated by a single blank line. The server then responds with:

```
"OK" CRLF
```

Retrieve Session

To retrieve data for an existing session the application sends a line of the form:

```
"get " appid " " sesid CRLF
```

If the session exists in the server it will respond with the following line:

```
"OK - session follows" CRLF
```

The session data saved previously will then be sent terminated by a single blank line.

Delete Session

To delete an existing session the application sends a line of the form:

```
"del " appid " " sesid CRLF
```

If the session exists it will be deleted and the server will respond with the following line:

```
"OK" CRLF
```

Quit

To disconnect from the server the application sends a line of the form:

```
"quit" CRLF
```

The server will then close the connection.

6.9 Application Deployment Options

In all of the previous sections you will note that all of the programs used the `Request` class from the `albatross.cgiapp` module to deploy the application as a CGI script.

The choice of `Request` class determines how you wish to deploy your application. Albatross supplies a number of pre-built Request implementations suited to various deployment methods. You should import the Request method from the appropriate module:

Deployment Method	Request Module
CGI	albatross.cgiapp
mod_python	albatross.apacheapp
FastCGI_python	albatross.fcgiapp
Stand-alone Python HTTP server	albatross.httpdapp

By placing all deployment dependencies in a `Request` class you are able to change deployment method with only minimal changes to your application mainline code. You could, for instance, carry out your initial development as a stand-alone Python HTTP server, where debugging is easier, and final deployment as FastCGI.

You could also develop your own `Request` class to deploy an Albatross application in other ways, such as using the Medusa web server (http://www.amk.ca/python/code/medusa.html), or to provide a `Request` class which for performing unit tests on your application.

The chapter on mod_python contains an example where the popview application is changed from CGI to `mod_python`.

6.9.1 `CGI` Deployment

The `albatross.cgiapp` module contains a `Request` class to allow you to deploy your application using `CGI`.

CGI is the simplest and most common application deployment scheme. The application is started afresh by your web server to service each client request, and is passed client input via the command line and stdin, and returns it's output via stdout.

An example of a CGI application:

```
#!/usr/bin/python
from albatross.cgiapp import Request

class Application(...):
    ...

app = Application()
if __name__ == '__main__':
    app.run(Request())
```

6.9.2 `mod_python` Deployment

The `albatross.apacheapp` module contains a `Request` class to allow you to deploy your application using `mod_python` [2].

In the following example, we change the popview application from CGI to `mod_python`. The complete sample program is contained in the `samples/popview5` directory. Use the `install.py` script to install the sample.

```
cd samples/popview5
python install.py
```

The new `popview.py` mainline follows.

```
from albatross import ModularSessionApp, SessionAppContext
from albatross.apacheapp import Request
import popviewlib


class AppContext(SessionAppContext):

    def open_mbox(self):
```

[2] For more information on `mod_python`, including installation instructions, see http://www.modpython.org/.

```
        if hasattr(self.locals, 'mbox'):
            return
        self.locals.mbox = popviewlib.Mbox(self.locals.username, self.locals.passwd)

    def read_msg(self):
        if hasattr(self.locals, 'msg'):
            return
        self.locals.msg = self.locals.mbox[int(self.locals.msgnum) - 1]
        self.locals.msg.read_body()


class App(ModularSessionApp):

    def __init__(self):
        ModularSessionApp.__init__(self,
                                   base_url='popview.py',
                                   module_path='-=-install_dir-=-',
                                   template_path='-=-install_dir-=-',
                                   start_page='login',
                                   secret='-=-secret-=-',
                                   session_appid='popview5')

    def create_context(self):
        return AppContext(self)


app = App()


def handler(req):
    return app.run(Request(req))
```

The `handler()` function is called by `mod_python` when a browser request is received that must be handled by the program.

You also need to create a `.htaccess` file to tell Apache to run the application using `mod_python`.

```
DirectoryIndex popview.py
SetHandler python-program
PythonHandler popview
```

Assuming you install the popview sample below the `/var/www` directory you will need configure Apache settings for the `/var/www/alsamp` directory:

```
<Directory /var/www/alsamp/>
    AllowOverride FileInfo Indexes
    Order allow,deny
    Allow from all
</Directory>
```

6.9.3 `FastCGI` Deployment

The `albatross.fcgiapp` module contains a `Request` class to allow you to deploy your application using `FastCGI` [3].

Applications deployed via CGI often perform poorly under load, because the application is started afresh to service each client request, and the start-up time can account for a significant proportion of request service time. `FastCGI` attempts to address this by turning the application into a persistent server that can handle many client requests.

[3] For more information on `FastCGI`, including installation instructions, see http://www.fastcgi.com/.

Unlike mod_python, where applications run within the web server, FastCGI applications communicate with the web server via a platform-independent socket protocol. This improves security and the resilience of the web service.

To deploy your application via FastCGI and Apache, you need to configure Apache to load mod_fastcgi.so, configure it to start your script as a FastCGI server, and use the albatross.fcgiapp Request class in your application. As an example of Apache configuration:

```
LoadModule fastcgi_module /usr/lib/apache/1.3/mod_fastcgi.so

<IfModule mod_fastcgi.c>
    <Directory /usr/lib/cgi-bin/application/>
        AddHandler fastcgi-script py
        Options +ExecCGI
    </Directory>
</IfModule>
```

And the application main-line:

```
#!/usr/bin/python
from albatross.fcgiapp import Request, running

class Application(...):
    ...

app = Application()
if __name__ == '__main__':
    while running():
        app.run(Request())
```

6.9.4 Stand-alone Python HTTP Server Deployment

The standard Python libraries provide a pure-Python HTTP server in the BaseHTTPServer module. Code contributed by Matt Goodall allows you to deploy your Albatross applications as stand-alone scripts using this module to service HTTP requests.

Unlike the other Request classes, applications deployed via the stand-alone Python HTTP server do not require you to instantiate the Request class directly. Instead, the al-httpd script imports your application (specifically, the script that instantiates the application object), and starts the HTTP server.

Currently, applications deployed via the stand-alone http server are single threaded, meaning that other requests are blocked while the current request is being serviced. In many cases this is not a problem as the requests are handled quickly, but if your application takes a significant amount of time to generate it's results, you may want to consider other deployment options for production use.

The stand-alone http server makes it particularly easy to deploy Albatross applications, and is a great way to debug applications without the complications that mod_python and FastCGI necessarily entail.

Most of the Albatross samples can be run under the stand-alone server:

```
$ cd samples/tree2
$ al-httpd tree.app 8080 /alsamp/images ../images/
```

6.10 Albatross Exceptions

exception AlbatrossError
An abstract base class all Albatross exceptions inherit from.

exception `UserError`
: Raised on abnormal input from the user. All current use of this exception is through the `SecurityError` subclass.

exception `ApplicationError`
: Raised on invalid Albatross use by the application, such as attempting to set a response header after the headers have been sent to the client. Template errors are also instances of this exception.

exception `InternalError`
: Raised if Albatross detects an internal error (bug).

exception `ServerError`
: Raised on difficulties communicating with the session server or errors reading server-side session files.

exception `SecurityError`
: A subclass of `UserError`, this exception is raised when Albatross detects potentially hostile client activity.

exception `TemplateLoadError`
: A subclass of `ApplicationError`, this exception is raised if a template cannot be loaded.

exception `SessionExpired`
: A subclass of `UserError`, this exception is raised when a client attempts to submit form results against a session that has expired.

CHAPTER

SEVEN

EXTENSIONS

Extensions add optional functionality to Albatross. We suggest you gain some experience with the core Albatross functionality before tackling the topics covered here.

7.1 Albatross Forms Guide

Albatross Forms provides support for developing Albatross applications which gather data from the user, validate it, and then return the user's data to the app. Much of this work is mechanical and it is tedious and error prone writing the same code on different pages in the application.

Using Forms lets the developer organise the presentation of related data on a web page programmatically. The Forms support handles the basic layout, type conversions and validation as the user interacts with the form. By using centrally defined data types, presentation, validation and error reporting can be done consistently and modified easily. It has been our experience that development is much faster and lots of code can be removed from the Albatross templates where it's hard to read, difficult to test and gets mucked up by web designers using WYSIWYG design tools.

Albatross Forms is designed to use CSS to change the layout of the forms when they are displayed rather than encoding the HTML into the form tags. This consolidates the web site's presentation and makes it easier to change the presentation globally.

7.1.1 Concepts

There are three main concepts that sit behind the Albatross Forms implementation:

- **Field**

  A data input field. It can format its output and validate its input. It contains a copy of the value so that the user can edit it without needing to maintain a separate copy in the application.

- **Fieldset**

  Groups together a list of Fields and renders them in a table. It is conceptually related the HTML fieldset tag which groups related input fields together. Fieldsets are intended to only hold data fields. If you try to insert a Button into a Fieldset it's not likely to work, in part because Fieldset expects that each Field member will respond to certain methods, and in part because notionally is that buttons apply actions to all the fields in the fieldset.

- **Form**

  Manages the all of the fields in the form. Most interactions in the application are with Form instances: they coordinate loading values from model objects (typically attributes of classes) into the Fields, organise rendering and updating the values from the browser, validation, and storing the values back into the model objects.

In practice, the developer will assemble a `Form` instance containing one or more `Field` instances and place this `Form` into `ctx.locals` (optionally fetching field values from an associated data or *Model* class via the `load()` method).

The developer then refers to this `Form` via a new `<alx-form>` tag in the page template (note that the `<alx-form>` must still be contained within an `<al-form>` tag in the page template). When the template is executed, the `Form` will be rendered to an HTML table containing appropriate inputs (including any values associated with the `Fields`).

When the user subsequently submits their responses, the developer will call the `Form` instance `merge()` method from the `page_process()` method and the user values will be merged back to the associated data storage class (or *Model*).

7.1.2 Getting started

You need to have a version of Albatross which has the Albatross Forms support included (or have installed it by hand yourself). You can quickly test whether it is present by running:

```
>>> from albatross.ext import form
```

If it’s missing, you’ll see an import error.

Registering the <alx-form> tag

In each Albatross application, there is a point at which an `App` subclass is instantiated (usually in app.py or app.cgi or wherever the main entry point of your application is). This instance needs to be told about the new `<alx-form>` tag. This is done with code that looks something like this:

```
import albatross
from albatross.ext import form
...
if __name__ == '__main__':
    app = albatross.SimpleApp()
    app.register_tagclasses(*form.tag_classes)
    app.run(Request())
```

Alternatively, if you subclass one of the Albatross application classes, you can register the new tags in your subclass’s constructor method (__init__):

```
import albatross
from albatross.ext import form
...
class Application(albatross.SimpleApp):

    def __init__(self, *args):
        albatross.SimpleApp.__init__(self, *args)
        self.register_tagclasses(*form.tag_classes)
```

7.1.3 A simple example

Here is a simple example of how we could use Albatross Forms to collect a username and password from the user.

We need to define a model class to hold the data:

```
import pwd, crypt

class User:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def is_password_valid(self):
        try:
            pw = pwd.getpwnam(self.username)
        except KeyError:
            return False
        return (crypt.crypt(self.password, pw.pw_passwd) == pw.pw_passwd)
```

Next, we need to define a form to display the fields:

```
from albatross.ext.form import *

class LoginForm(FieldsetForm):

    def __init__(self, user):
        fields = (
            TextField('Username', 'username'),
            PasswordField('Password', 'password'),
        )
        fieldsets = (Fieldset(fields), )
        buttons = Buttons((
            Button('Login', 'login'),
        ))
        FieldsetForm.__init__(self, 'User login', fieldsets, buttons=buttons)
        self.load(user)
```

We need to create an instance of the Login model and maintain that so that any captured data is retained. In our login.py, we use:

```
def page_enter(ctx):
    if not ctx.has_value('user'):
        ctx.locals.user = User('', '')
        ctx.add_session_vars('user')
        ctx.locals.login_form = LoginForm(ctx.locals.user)
        ctx.add_session_vars('login_form')
    ctx.locals.login_error = ''
```

In login.html, to display the form to the user we use:

```
<al-form method="post">
    <alx-form name="login_form" errors />
    <al-expr expr="login_error" />
</al-form>
```

When the user presses the “Login” button, it will come back to our page_process method in login.py. We check if the username and password are correct and punt them into the application proper (via the “search” page) or tell them they’ve got it wrong:

```
def page_process(ctx):
    if ctx.req_equals('login'):
        # nothing to validate
        ctx.locals.login_form.merge(ctx.locals.user)
        if ctx.locals.user.is_password_valid():
            ctx.redirect('search')
        else:
            ctx.locals.login_error = 'Login incorrect'
```

7.1.4 Flow of Control

The flow of control through Albatross Forms is tied in with Albatross's flow of control.

A common mistake is to reinitialise the form from the model part way through the user's interaction with the page (ie, before they've saved it). It winds up losing any changes that the user has made on the form. The lesson here is that the form should only be loaded from the model once when the user starts interacting with it; don't reload it on each page refresh.

1. **Constructor**

   Create the form itself.

2. **Load values from model**

   This is often done in the form subclass constructor method (__init__).

3. **Display the form**

   Render the form to the web page, using <alx-form name="model_form" errors> in your Albatross template for the page.

4. **Validate**

   Check that the data that the user entered is correct. The call to validate will raise a FormValidationError exception.

5. **Merge**

   Update the data class (*Model*) with the data fields collected from the form.

7.1.5 Field types

Albatross Forms defines a number of standard fields. You can also add your own, subclassing the standard fields to add validation or type-casting. The standard fields are:

TextField

A normal text input. Corresponds to the `<input type="text">` tag.

PasswordField

Same as a text field but it doesn't display the characters as the user enters them. Corresponds to the `<input type="password">` tag.

StaticField

A TextField with static content.

TextArea

Corresponds to the `<textarea>` tag.

IntegerField

An integral value, `get_value()` will return an `int` type, non-integer values will result in a `FieldValidationError` on `validate()`.

FloatField

A floating point value, `get_value()` will return a `float` type, non-floating point values will result in a `FieldValidationError` on `validate()`.

Checkbox

Renders as <input type="checkbox">, `get_value()` will return a `bool` type.

SelectField

Returns one of the values listed in the `(value, display_value)` list passed to the constructor. If the value can be converted to an `int`, it will be; otherwise it will be returned as a `str`.

RadioField

Returns one of the values listed in the `(value, display_value)` list passed to the constructor. If the value can be converted to an `int`, it will be; otherwise it will be returned as a `str`.

FileField (not implemented yet)

Internal storage within a field

The interaction of the field with the browser is complicated because the browser requires and returns string values, which results in the loss of type information for the objects that the fields wrap.

To manage this, the field class notes when the field is rendered to the browser and converts the value to a string using the object's `get_display_value()` method. When the values are posted back to the form by the browser, they are returned as strings. If you need to access the value stored in a field, use the `get_value()` method which tracks the type of the stored representation and does a conversion (using `get_merge_value()`) when appropriate.

7.1.6 A more complex example

Here is a rather more complex example that uses a number of different input types.

Here's the model:

```
class User:
    def __init__(self):
        self.name = ''
        self.an_int = 0
        self.a_float = 0.0
        self.country = 0
        self.password = ''
        self.active = False
```

The form that describes how we want it laid out:

```
# Slightly abridged list of all the countries in the world.
country_names = (
    'Australia',     'Belgium',      'Cuba',
    'Greenland',     'Madagascar',   'Netherlands',
    'Switzerland',   'Uzbekistan',   'Zimbabwe'
)
country_menu = [e for e in enumerate(country_names)]
fields = (
    TextField('Name', 'name', required=True),
    IntegerField('Integer', 'an_int'),
    FloatField('Float', 'a_float'),
    SelectField('Country', country_menu, 'country'),
    PasswordField('Password', 'password',
                  required=True),
    Checkbox('Active', 'active'),
)
buttons = Buttons((
    Button('Save', 'save'),
    Button('Cancel', 'cancel'),
))
fieldsets = (Fieldset(fields), )
```

```
ctx.locals.test_form = FieldsetForm('User details',
                                    fieldsets,
                                    buttons=buttons)
```

Render the form on the page:

```
<al-form method="POST">
   <div class="alxform">
     <alx-form name="test_form" errors />
   </div>
 </al-form>
```

To render the form as static report:

```
<div class="alxform">
  <alx-form name="test_form" static />
</div>
```

In the forms.py file, the code looks like:

```
def page_enter(ctx):
    if not ctx.has_value('test_form'):
        ctx.locals.description = User()
        ctx.add_session_vars('description')
        ctx.locals.test_form = test_form # see above

def page_display(ctx):
    ctx.run_template('forms.html')

def page_process(ctx):
    if ctx.req_equals('save'):
        try:
            ctx.locals.test_form.validate()
        except FormValidationError, e:
            ctx.locals.test_form.set_disabled(False)
            return
        ctx.locals.test_form.set_disabled(True)
        ctx.locals.test_form.merge(ctx.locals.description)
    elif ctx.req_equals('reset'):
        ctx.locals.test_form.clear()
        ctx.locals.test_form.set_disabled(False)
    elif ctx.req_equals('cancel'):
        if ctx.locals.test_form.disabled:
            ctx.locals.test_form.set_disabled(False)
        else:
            ctx.locals.test_form.clear()
            ctx.redirect('main')
```

7.1.7 Customising Fields

Here is an example of how we created a `MoneyField` that knows how to validate a currency value. Both parser and formatter are modules that we wrote to convert between formats. Our modules deal in dollars and cents but that's hidden from the application code.

Note that the parser needs to be able to parse the output of the formatter: the field will be initialised with the formatter's output when it is rendered. It is reasonable to expect the parser to accept "$5.50" if that is the format that the application is presenting to the user.

```
import parser, formatter

class MoneyField(FloatField):

    def validate(self, form, s):
        s = s.strip()
        if not self.required and not s:
            return
        try:
            parser.money(s)
        except ValueError, e:
            raise FieldValidationError('Invalid value "%s" for money' % s)

    def get_merge_value(self, s):
        s = s.strip()
        if not self.required and not s:
            return
        return parser.money(s)

    def get_display_value(self, ctx, form):
        return formatter.money(self.get_value())
```

7.1.8 Attaching buttons to a form

It's common to have buttons at the bottom of a form even if they just say "Save" and "Cancel". This is supported in Albatross Forms by adding an optional keyword arg when creating the Form object, for example:

```
from albatross.ext.form import *

class LoginForm(FieldsetForm):

    def __init__(self, user):
        fields = (
            TextField('Username', 'username'),
            PasswordField('Password', 'password'),
        )
        fieldsets = (Fieldset(fields), )
        buttons = Buttons((
            Button('Login', 'login'),
        ))
        FieldsetForm.__init__(self, 'User login', fieldsets, buttons=buttons)

        self.load(user)
```

The `Buttons` class takes a list of `Button` instances in its constructor and displays buttons in the bottom right hand corner of the form display.

You check for the buttons being clicked by the user in the usual Albatross way in *page_process*:

```
def page_process(ctx):
    if ctx.req_equals('save'):
        ...
```

7.1.9 Table support

Rendering tables using Albatross Forms is relatively straightforward, using them for input is no harder.

Table support revolves around two classes: `IteratorTable` and `IteratorTableRow`.

IteratorTable acts as a field in a form in which the table is rendered. The first argument to the IteratorTable constructor is the name of the attribute in the class in which the the table is stored. This is necessary so that Albatross can navigate through the form's fields to update the values from the user's browser. It's a little arcane but it isn't too bad.

IteratorTableRow should be subclassed within the application to render each row in turn.

The IteratorTable class steps through the list of objects that it's passed and calls the IteratorTableRow subclass that's specified with each object in turn. Each of these is responsible for rendering a single row.

When the IteratorTable is rendered, it will display the header columns (if specified) and then ask each row to render itself.

Here's an example which should render a list of name, address and phone numbers in a table. First we define the model object:

```
class Entry:

    def __init__(self, name, address, phone):
        self.name = name
        self.address = address
        self.phone = phone
```

Now we'll define the components of the form to render a list of Entry instances:

```
class EntryTableRow(IteratorTableRow):

    def __init__(self, entry):
        cols = (
            Col((TextField('Name', 'name'), )),
            Col((TextField('Address', 'address'), )),
            Col((TextField('Phone', 'phone'), )),
        )
        IteratorTableRow.__init__(self, cols)

        self.load(entry)


class EntryTableForm(Form):

    def __init__(self, entries):
        headers = Row((
            HeaderCol((Label('Name'), )),
            HeaderCol((Label('Address'), )),
            HeaderCol((Label('Phone'), )),
        ))
        self.table = IteratorTable('table', headers, EntryTableRow, entries,
                                   html_attrs={'width': '100%'})
        Form.__init__(self, 'Address book', (self.table, ))
```

To create the form:

```
def page_enter(ctx):
    entries = [
        Entry('Ben Golding',
              'Object Craft, 123/100 Elizabeth St, Melbourne Vic 3000',
              '+61 3 9654-9099'),
        Entry('Dave Cole',
              'Object Craft, 123/100 Elizabeth St, Melbourne Vic 3000',
              '+61 3 9654-9099'),
    ]
    if not ctx.has_value('entry_table_form'):
        ctx.locals.entry_table_form = EntryTableForm(entries)
        ctx.add_session_vars('entry_table_form')
```

Rendering the list just requires:

```
<al-form method="post">
    <div class="alxform">
        <alx-form name="entry_table_form" static />
    </div>
</al-form>
```

To support editing the fields, you would change how it renders using:

```
<al-form method="post">
    <div class="alxform">
        <alx-form name="entry_table_form" errors />
    </div>
</al-form>
```

When the user has made changes, your page_process method can pick up the changes using:

```
def page_process(ctx):
    if ctx.req_equals('save'):
        ctx.locals.entry_table_form.merge(ctx.locals.entries)
        save(ctx.locals.entries)
```

Paginating a table

To add pagination to a table, you need to specify a pagesize and the rest of the work is done for you.

```
<al-form method="post">
    <div class="alxform">
        <alx-form name="entry_table_form" errors pagesize="15" />
    </div>
</al-form>
```

Currently, the simple paginator only emits "Prev" and "Next" links at the bottom of the table. It's straightforward to change the style of those links by subclassing the `IteratorTable` class: pass in an instance of your own custom subclass of `PageSelectionDisplayBase` as the `page_selection_display` argument of `IteratorTable`'s constructor to render the page links differently.

The `pagesize` argument can also be specified when creating an `IteratorTable` instance. The argument specified overrides any that are used in the `<alx-form>` tag.

Adding rows to a table

The developer is responsible for keeping the form's idea of the number of rows in the table in sync with the rows in the model.

```
def page_process(ctx):
    ...
    if ctx.req_equals('add_entry'):
        entry = Entry('', '', '')
        ctx.locals.entries.append(entry)
        ctx.locals.entry_table_form.table.append(entry)
```

Note that the `IteratorTable.append()` method will call the row class with the model data that's specified in the constructor.

Adding heterogenous rows to a table

If you were displaying a series of rows each of which was a product, at the end of the table it would be great to display a total entry for all of the included lines. In this example, we use `append_row()` to append a pre-formatted row (ie, an `IteratorTableRow` subclass instance) to the table.

Note that while this works when rendering a form, I don't think it will work if the form is used for input.

```
class ProductTableRow(IteratorTableRow):

    def __init__(self, product):
        cols = (
            Col((TextField('Code', 'code'), )),
            Col((TextField('Name', 'name'), )),
            Col((FloatField('Cost', 'cost'), ),
                 html_attrs={'class': 'number-right'}),
        )
        IteratorTableRow.__init__(self, cols)
        self.load(product)


class ProductTotalRow(IteratorTableRow):

    def __init__(self, products):
        cols = (
            Col((Label(''), )),
            Col((Label('Total'), )),
            Col((FloatField('Total', value=products.total_amount(), static=True), ),
                 html_attrs={'class': 'number-right'}),
        )
        IteratorTableRow.__init__(self, cols)


class ProductTableForm(FieldsetForm):

    def __init__(self, products):
        headers = Row((
            HeaderCol((Label('Product code'), )),
            HeaderCol((Label('Product name'), )),
            HeaderCol((Label('Cost'), ),
                      html_attrs={'class': 'number-right'}),
        ))
        self.table = IteratorTable('table', headers, ProductTableRow, products,
                                   html_attrs={'width': '100%'})
        self.table.append_row(ProductTotalRow(products))
        buttons = Buttons((
            Button('save', 'Save'),
            Button('cancel', 'Cancel'),
        ))
        FieldsetForm.__init__(self, 'Product table', (self.table, ),
                              buttons=buttons)
```

Deleting rows from a table

When deleting rows from a table, I normally put a check box next to each of the rows and include a "delete selected" button so that the user can delete multiple rows at once.

In the table row constructor, I poke an is_selected value into the model object as a placeholder for the selected check box. I feel like this is impolite but it works very effectively.

```
class EntryTableRow(IteratorTableRow):

    def __init__(self, entry):
        entry.is_selected = False          # placeholder
        cols = (
            Col((Checkbox('Selected', 'is_selected'), )),
            Col((TextField('Name', 'name'), )),
            Col((TextField('Address', 'address'), )),
            Col((TextField('Phone', 'phone'), )),
        )
        IteratorTableRow.__init__(self, cols)

        self.load(entry)
```

When processing the request, I step through each list element in the model list in in sync with each child in the table form and delete both of them when the checkbox is selected.

```
def page_process(ctx):
    ...
    elif ctx.req_equals('delete_selected'):
        for entry, entry_form_child in zip(ctx.locals.entries,
                                           ctx.locals.entry_table_form.table.children):
            is_selected_field = entry_form_child.get_field('Selected')
            if is_selected_field.get_value():
                ctx.locals.entries.remove(entry)
                ctx.locals.entry_table_form.table.remove(entry_form_child)
```

7.1.10 Querying fields before merge

Sometimes it can be really nice (read: improve the user experience) to be able to query some fields without merging the value from the form back into an object. An example is when a user changes a input select field in a form which has an attached value. In an app we have, a user has a select list of product numbers. We make the app update the product's name on the page when the product changes.

Note that we are treading on thin ice here: if the user has not entered a valid value for the field, when `get_value()` tries to convert it to the appropriate type, it will raise an exception that will need to be dealt with. It is (generally) safe to reference values from a `TextField` or a `SelectField` though.

```
class ProductFieldsetForm(FieldsetForm):

    products_menu = product_factory.all_products_menu()

    def __init__(self, product):
        self.product_id = product.product_id
        elements = (
            Fieldset((
                SelectField('Product code', self.products_menu, 'product_id',
                            html_attrs={'onchange': 'javascript:product_form.submit()'}),
                StaticField('Product', product.name),
            )),
        )
        FieldsetForm.__init__(self, 'Product selection', elements)
        self.load(product)

    def update(self):
        product_code_field = self.get_field('Product code')
        if self.product_id != product_code_field.get_value():
            self.product_id = product_code_field.get_value()
            product = product_factory.product_with_product_id(self.product_id)
            product_field = self.get_field('Product')
            product_field.set_value(product.name)
```

The interesting part here is where we query the form for a specific field using its `get_field()` method.

We use the little bit of Javascript to make a change to the select list force an update to the page. That means having to name the form when we render the page:

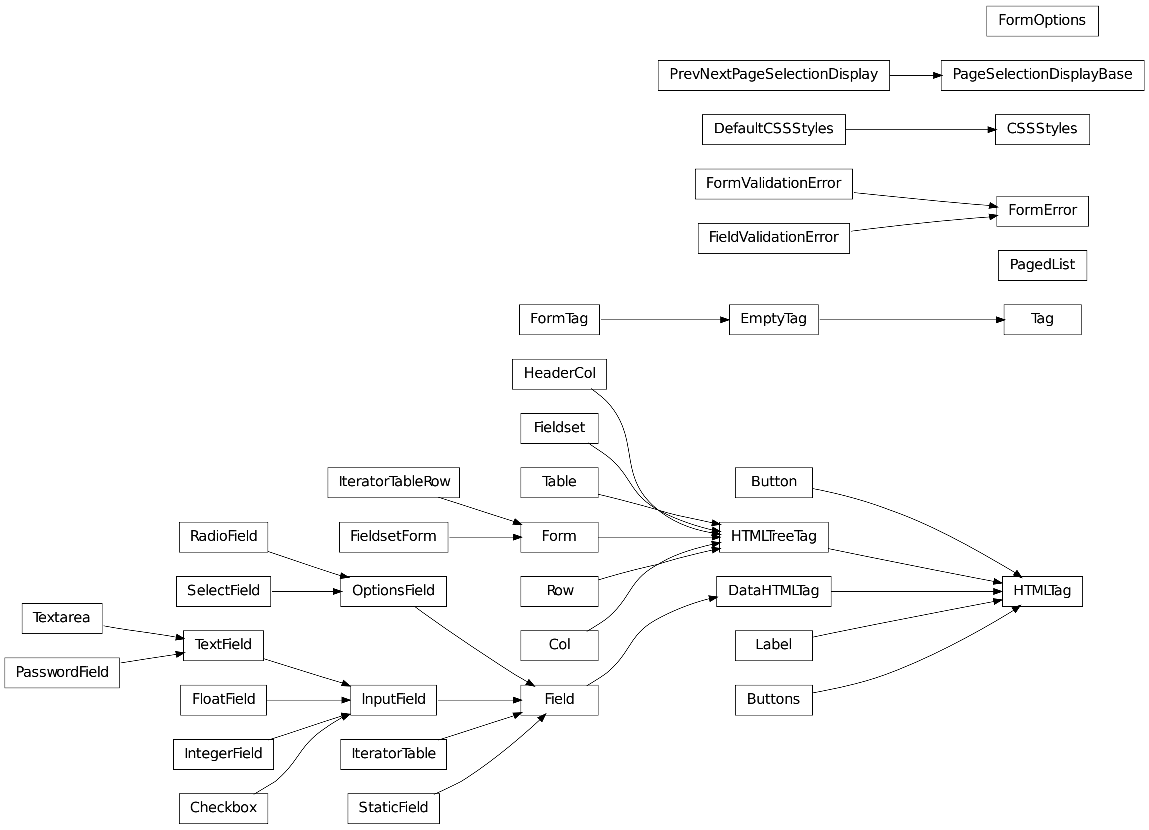
```
<al-form method="post" name="product_form">
    <alx-form name="product_form_form" />
</al-form>
```

In the app itself, we use:

```
def page_enter(ctx):
    if not ctx.has_value('product_form'):
        ctx.locals.product_form = ProductForm(ctx.locals.product)
        ctx.add_session_vars('product_form')

def page_process(ctx):
    ctx.locals.product_form.update()      # update product name if product changed
    if ctx.req_equals('save'):
        ...
```

7.2 Albatross Forms Reference



class **`Button`** (*value, name, html_attrs=None*)
Bases: `albatross.ext.form.HTMLTag`

An HTML *submit* button. All buttons in a `Form` should be collected in a single `Buttons` instance which is passed to the constructor of the `Form`.

- *value* is written on the face of the button when displayed.
- *name* is returned from the browser and can be tested in the application using *ctx.req_equals('name')*.

class **`Buttons`** (*buttons, html_attrs=None*)
Bases: `albatross.ext.form.HTMLTag`

A collection of `Button` instances.

class **`Checkbox`** (*name, attr, merge_obj=None, value=None, html_attrs=None, required=False, static=False*)
Bases: `albatross.ext.form.InputField`

An HTML <input type="checkbox"> control. *self.value* is always valid and the merged value is boolean.

class **`Col`** (*children, html_attrs=None*)
Bases: `albatross.ext.form.HTMLTreeTag`

HTML *TD* tag.

class **`DataHTMLTag`** (*attr, value=None, html_attrs=None*)
Bases: `albatross.ext.form.HTMLTag`

Output HTML for a value which which can be loaded from another object.

- *attr* is the name of the attribute in the model manipulated by `load()` and `merge()`.

`get_display_value`(*ctx, form*)
Override this method to provide any required formatting or modification of the displayed value.

`get_merge_value`(*s*)
This should return the correctly typed object to update the field in the original object. Override this if you are using a non-string value.

`get_value`()
Return the converted value in this field

class **`DefaultCSSStyles`**()
Bases: `albatross.ext.form.CSSStyles`

A collection of named CSS styles used in the HTML output of `HTMLTag` instances.

Each `Form` has a reference to a single instances of this class - `field.styles`.

Additional styles can be added to the default instance after the `Form` is created or an existing instance can be specified when the `Form` is created using the *styles* argument.

The default style names are:

- buttons
- legend
- label
- field
- field-error
- field-invalid,
- value
- value-static

class **`Field`**(*name, attr, merge_obj=None, value=None, html_attrs=None, required=False, static=False*)
Bases: `albatross.ext.form.DataHTMLTag`

Base class for input fields.

- *name* is the display name tyically shown to the left of the input control in the generated HTML.
- Specify *merge_obj* if this `Field` is loaded/merged from a model object other than the one used for the `form` (this should usually be left as *None*).
- Set *required* to *True* if a value must be entered for this `Field` (a `FieldValidationError` will be raised on a call to `validate()` if the `Field` is empty).
- If *static* is *True* the `Field` is always displayed in static report style preventing user input.

Creating your own Field-subclasses

Several methods may be provided by custom subclasses as needed:

- ***get_display_value(self, ctx, form)*** should return the value stored in *self.value* to how you want it displayed in the browser. This is typically called during to_html and the value of *self.value* will be replaced with this after that point.
- ***get_merge_value(self, s)*** Should convert the string version of the value passed in and return a value of the appropriate type.
- ***validate(self, form)*** Validate the string version of the value stored in the field. The method should just return if it validates correctly; if not, raise `FormValidationError` with an appropriate error message.

  `Field`'s that are static or are members of a static form are not validated.

•***to_html(self, ctx, form)*** Generate custom HTML for this tag. Use:

form.write_content(ctx, '<your html here>')

to write your HTML to the output stream.

`get_merge_value`(*value*)
Override `get_merge_value()` where the value received from Albatross on a form submission should be checked or modified before being stored.

`input_name_for_form`(*form*)
Create a unique string representation of the field to be used as an Albatross input name.

`load`(*form_load_obj*)
Load *self.value* from a named attribute in the model.

`merge`(*form_merge_obj=None*)
Merge value back into the model.

`to_html`(*ctx, form*)
Field subclasses must provide different output for static, enabled and disabled states and may output validation error messages.

The default implementation of `write_static_html()` and `write_errors_html()` should suffice for most cases. A typical subclass will override `write_form_html()` to provide the HTML ouput for the input controls required (text input, checkbox, etc.)

`validate`(*form, s*)
Subclasses should use this method to validate *s* after form submission. The method should raise a `FieldValidationError` for invalid values.

`write_errors_html`(*ctx, form*)
Write the HTML representation of the `FieldValidationError` for this `Field`. Called during HTML generation if this `Field` raised an exception during `Form` validation.

`write_form_html`(*ctx, form*)
Write the interactive form control HTML representation of the field.

`write_static_html`(*ctx, form*)
Write a static HTML representation of the field. This method is called by `to_html()` if *self.static* is True of if the whole `Form` is being statically rendered.

exception `FieldValidationError`
Bases: `albatross.ext.form.FormError`

Raised by `Field` subclasses if `validate()` fails.

class `Fieldset`(*children, html_attrs=None*)
Bases: `albatross.ext.form.HTMLTreeTag`

Container for a collection of `Field` instances.

Writes a table of fields. Note that the <fieldset> HTML tags are intentionally provided by the `FieldsetForm` and not this class.

class `FieldsetForm`(*legend, children, buttons=None, styles=None, html_attrs=None*)
Bases: `albatross.ext.form.Form`

A `Form` subclass which wraps the HTML output in a *<FIELDSET>* tag.

class `FloatField`(*name, attr, merge_obj=None, value=None, html_attrs=None, required=False, static=False*)
Bases: `albatross.ext.form.InputField`

A `TextField` which validates and merges float values.

class `Form`(*legend, children, buttons=None, styles=None, html_attrs=None*)
Bases: `albatross.ext.form.HTMLTreeTag`

Encapsulates an HTML form and its input fields.

- *legend* is displayed as the title of the form.
- *children* is a list of `HTMLTag` subclasses including all the `Form` fields and markup fields for layout.
- *buttons* is an optional `Buttons` instance.
- *styles* is an optional `CSSStyles` instance. If not specified a `DefaultCSSStyles` instance is created.

clear()
: Clear all field values, set form and all fields to valid state, clear disabled and set edit mode to `FORM_CREATE`.

clear_errors()
: Clear errors and set all fields to valid.

load(*load_obj*)
: Load field values from *load_obj*.

merge(*merge_obj=None*)
: Merge all field values to *merge_obj*.

run(*ctx, name, opts*)
: Called by alx-form tag handler to render the form. This method sets the correct internal state for rendering, resets indentation and registers input fields before calling `to_html()`.

set_disabled(*is_disabled*)
: Disable or enable data entry on all fields on this form.

to_html(*ctx*)
: Render all fields and buttons as HTML. This is an internal method. To render the form use `run()`.

validate()
: Call `validate()` on each field. Sets *self.valid* to False and raises `FormValidationError` if any fields are invalid. Validation exceptions are stored in *self.validation_errors* and are cleared by calling `clear()`, `clear_errors()` or on the next call to `validate()`.

  Static fields or fields of a static form are not validated.

write_content(*ctx, content, indent=None*)
: A wrapper for `ctx.write_content()` which provides indentation to assist in generating human readable HTML output.

  Valid values for *indent* are:

  - INDENT
  - DEDENT
  - POST_INDENT
  - POST_DEDENT

  All other values are ignored.

exception FormError
: Bases: `exceptions.Exception`

  Base class for all exceptions raised by the `albatross.ext.form` module.

class FormOptions(*show_errors=False, static=False, pagesize=None*)
: Collect the attributes we're interested in from the alx-form tag.

exception FormValidationError
: Bases: `albatross.ext.form.FormError`

  Raised by `Form` subclasses validation fails for one or more fields.

  - *errors* is a dictionary of `FieldValidationError` instances keyed by `Field` instance.

class `HTMLTag`(*html_attrs=None*)
Base class for any class which provides HTML output. Arbitrary HTML attributes can be specified as *html_attrs*.

class `HTMLTreeTag`(*children, html_attrs=None*)
Bases: `albatross.ext.form.HTMLTag`

An `HTMLTag` with children.

class `HeaderCol`(*children, html_attrs=None*)
Bases: `albatross.ext.form.HTMLTreeTag`

HTML *TH* tag.

class `InputField`(*name, attr, merge_obj=None, value=None, html_attrs=None, required=False, static=False*)
Bases: `albatross.ext.form.Field`

Base class for generating an HTML <input> tag. Requires that the instance has a *self.type* which is the type of the HTML input, ie, it will generate <input type="(*self.type*)" ...>.

Subclasses are expected to maintain *self.value* as the correct type (eg, *string* (for text), *int*, *datetime*, etc) and to do validation of the input as required. *Self.value* can also contain invalid strings if the user has partly edited a field. Calling *merge* or *get_value* before *validate* has succeeded may raise an exception.

The conversion of the internal value to the display value is complicated because Albatross sets the value of an empty field to None so that needs to be handled or all empty text fields in a form are converted to "None".

class `IntegerField`(*name, attr, merge_obj=None, value=None, html_attrs=None, required=False, static=False*)
Bases: `albatross.ext.form.InputField`

A `InputField` which validates and merges integer values.

class `IteratorTable`(*table_attr, header_row, row_class, content_list, html_attrs=None, pagesize=None, page_selection_display=<albatross.ext.form.PrevNextPageSelectionDisplay instance at 0xa78e50c>*)
Bases: `albatross.ext.form.Field`

Create an HTML table from a list of objects.

- *table_attr* is the name of the instance var in the class which creates this object. This is used when directing Albatross to update the form values.
- *header_row* is a `Row` of `HeaderCol` instances (usually) which are used to put a header on the table. An empty list or None will suppress any headers.
- *row_class* is a subclass of `IteratorTableRow`. It is used to render each row in the table. It will be used to contruct each row by being instantiated with each element of *content_list* in turn.

  If you need pass the row class constructor some extra arguments from the table constructor, assign them to instance variables and make the *row_class* a bound method to a method that marshalls the arguments before calling the row constructor.
- *content_list* is the content of the table to be displayed.

`append`(*content_item*)
Append a new row of data to the table

`append_row`(*row*)
Append an IteratorTableRow instance to the table

`goto_page`(*page*)
Jump to specified page. If page == -1, go to last page.

class `IteratorTableRow`(*cols*)
Bases: `albatross.ext.form.Form`

When using an IteratorTable, the class that's used to display each row must be a subclass of IteratorTableRow so that the rendering and validation is performed correctly. This is checked for in the `IteratorTable` constructor.

Each row in a table is actually treated as a separate sub-`Form`. This allows Albatross to traverse the table hierarchy when it's updating the field values.

`to_html`(*ctx*)
: generates the row's content bracketed by <tr>/</tr>

class **`Label`**(*value, html_attrs=None*)
: Bases: `albatross.ext.form.HTMLTag`

  Simple displayed string value with no label.

class **`OptionsField`**(*name, options, attr, merge_obj=None, value=0, html_attrs=None, static=False*)
: Bases: `albatross.ext.form.Field`

  Base class to handle selection of a single value from a list of options, ie, for a select or radio list.

  `set_options`(*options*)
  : change the *options* displayed by the select field.

class **`PageSelectionDisplayBase`**()
: Base class for displaying page selection

class **`PasswordField`**(*name, attr, merge_obj=None, value=None, html_attrs=None, required=False, static=False*)
: Bases: `albatross.ext.form.TextField`

  A `TextField` with text input hidden by '*'

class **`RadioField`**(*name, options, attr, merge_obj=None, value=0, html_attrs=None, static=False*)
: Bases: `albatross.ext.form.OptionsField`

  Manage a radio list.

  - *options* is a list of tuples of (value, display value)

  When the field or the enclosing form is static, we just emit the selected option. (XXX I hope that's the right thing to do)

class **`Row`**(*children, html_attrs=None*)
: Bases: `albatross.ext.form.HTMLTreeTag`

  HTML *TR* tag.

class **`SelectField`**(*name, options, attr, merge_obj=None, value=0, html_attrs=None, static=False*)
: Bases: `albatross.ext.form.OptionsField`

  Manage a drop down list of options.

  - *options* is a list of tuples of (value, display value)

class **`StaticField`**(*name, value, html_attrs=None*)
: Bases: `albatross.ext.form.Field`

  String `Field` which never accepts user input.

class **`Table`**(*header_row, children, html_attrs=None*)
: Bases: `albatross.ext.form.HTMLTreeTag`

  HTML *TABLE* tag.

class **`TextField`**(*name, attr, merge_obj=None, value=None, html_attrs=None, required=False, static=False*)
: Bases: `albatross.ext.form.InputField`

  HTML <input type="text"> tag. Merged data is a string. If *self.required* is *True* `validate()` will raise a `FieldValidationError` if the input field is empty. No other validation or conversion is performed.

`get_merge_value`(*value*)
Empty input field is submitted as None - convert to empty string.

class **`Textarea`**(*name, attr, merge_obj=None, value=None, html_attrs=None, required=False, static=False*)
Bases: `albatross.ext.form.TextField`

An HTML *TEXTAREA* tag. Validate and merge rules are the same as `TextField`.

CHAPTER

EIGHT

TEMPLATES REFERENCE

Albatross provides an HTML templating system that applications use to provide the presentation layer.

The template parser uses regular expressions to locate all of the template tags. All text that is not recognised as an Albatross tag or associated trailing whitespace is passed unchanged through the parser. In practice this means that you can use the templating system for non-HTML files.

The parser regular expressions recognise all tags names that are prefixed by either “al-” or “alx-“. All standard Albatross tags use the “al-“. The “alx-” prefix is provided to ensure extension tag names do not clash with standard names.

For Albatross tags that enclose content you can use the XML empty tag syntax to indicate that there is no content. The parser transforms all tag and attribute names to lowercase and allows attributes on multiple lines. For example the following two constructs are identical.

```
<al-for iter="i" expr="seq" pagesize="10" prepare>
</al-for>

<al-for iter="i"
        expr="seq"
        pagesize="10"
        prepare/>
```

The parser handles attribute values enclosed with either single or double quotes. The enclosing quote character can be used in the attribute string if it is escaped by a backslash (“$\”). Attribute values can be broken over multiple lines.

8.1 Fake Application Harness

Some of the explanatory examples in this chapter require application functionality. The following fake application is used as an execution harness to drive the interactive examples.

```
import sys
import albatross


class Request:

    def get_uri(self):
        return 'http://www.com/fakeapp.py'

    def write_header(self, name, value):
        pass

    def end_headers(self):
        pass
    def write_content(self, data):
        sys.stdout.write(data)


app = albatross.SimpleApp(base_url='fakeapp.py',
                          template_path='.',
                          start_page='start',
                          secret='secret')

ctx = albatross.SessionAppContext(app)
ctx.set_request(Request())
```

8.2 Enhanced HTML Tags

Tags in this section are used in place of standard HTML tags to access values from the execution context.

All attributes that are not recognised by Albatross are passed without modification to the generated HTML.

8.2.1 `<al-form>`

Albatross browser request merging depends upon the functionality provided by the `<al-form>` tag. If you do no use this tag in applications then the standard request merging will not work.

The tag will automatically generate the HTML `<form> action` (*action="..." attribute*) and `enctype` (*enctype="..." attribute*) attributes as required.

If you are using an execution context that inherits from the `NameRecorderMixin` (nearly all do — see chapter *Prepackaged Application and Execution Context Classes*) then the execution context will raise a `ApplicationError` exception if multiple instances of some types of input tag with the same name are added to a form. The `list` attribute of the `<al-input>` tag is used indicate that multiple instances are intentional.

`action="..."` attribute

If you do not supply an `action` attribute the tag will generate one with based on the value returned by the `current_url()` method of the execution context.

```
>>> import albatross
>>> from fakeapp import ctx
>>> albatross.Template(ctx, '<magic>', '''
... <al-form whitespace>
...  <al-input type="text" name="name" whitespace>
... </al-form whitespace>
...
... <al-form action="http://there.com/" whitespace>
...  <al-input type="text" name="name" whitespace>
... </al-form whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<form action="/fakeapp.py">
 <input type="text" name="name" />
<div><input type="hidden" name="__albform__" value="eJzb9mXFmseCLDkMLy6Hzfx1N6dK5rJwA1NtIWMoS
" /></div>
</form>

<form action="http://there.com/">
 <input type="text" name="name" />
<div><input type="hidden" name="__albform__" value="eJzb9mXFmseCLDkMLy6Hzfx1N6dK5rJwA1NtIWMoS
```

```
" /></div>
</form>
```

Note that the generated `action` attribute is relative to the document root.

`enctype="..."` attribute

If you include any file input fields then the open tag will automatically supply an `enctype="multipart/form-data"` attribute.

```
>>> import albatross
>>> from fakeapp import ctx
>>> albatross.Template(ctx, '<magic>', '''
... <al-form whitespace>
...  <al-input type="text" name="name" whitespace>
...  <al-input type="file" name="data" whitespace>
... </al-form whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<form action="/fakeapp.py" enctype="multipart/form-data">
 <input type="text" name="name" />
 <input type="file" name="data" />
<div><input type="hidden" name="__albform__" value="eJzjECjXXD/Z7bjIvorcXnm3j10Inf83MNUWMmqEs
" /></div>
</form>
```

8.2.2 `<al-input>`

Albatross browser request merging depends upon the functionality provided by the `<al-input>` tag. If you do no use this tag in applications then the standard request merging will not work.

In the most common usage a `value` (*value="..." attribute*) attribute is generated by evaluating the `name` (*name="..." attribute*) attribute in the execution context. This can be overridden by supplying a `value` (*value="..." attribute*) or `expr` (*expr="..." attribute*) attribute.

There are a number of attributes that automatically generate the `name` attribute.

When merging browser requests the application object places the browser supplied value back into the execution context value referenced by the `name` attribute. The application request merging will not merge variable names prefixed by underscore. Use this to protect application values from browser modification.

`alias="..."` attribute

This attribute is ignored is any of the following attributes are present; `prevpage`, `nextpage` (*prevpage="..." and nextpage="..." attributes*), `treefold`, `treeselect`, or `treeellipsis` (*treeselect="...", treefold="..." and treeellipsis="..." attributes*).

The value of the `alias` attribute is passed to the `make_alias()` method of the execution context. The return value is then used as the generated `name` (*name="..." attribute*) attribute.

The execution context `make_alias()` method splits the `alias` attribute at the last '.' and resolves the left hand side to an object reference. The `albatross_alias()` method is then called on that object and the result is combined with the '.' and the right hand side of the of the `alias` attribute to produce the generated `name` attribute. The resolved object is entered in the the local namespace and the session using the name returned by the `albatross_alias()` method.

The template `samples/tree/tree1.html` contains an example of this method for generating a `name` attribute.

```
<al-tree iter="n" expr="tree">
 <al-for iter="c" expr="range(n.depth())">
  <al-value expr="n.line(c.value())" lookup="indent">
 </al-for>
 -<al-input type="checkbox" alias="n.value().selected">
  <al-value expr="n.value().name" whitespace="newline">
</al-tree>
```

Note that each node in the tree has a checkbox that controls whether or not the node is selected. When processing the `alias` attribute the toolkit isolates the left hand side (`n.value()`) which happens to be the current tree node of `TreeIterator` `n`. To generate the alias the `albatross_alias()` method of the current node is called. In `samples/tree/tree1.py` the implementation of that method looks like:

```
class Node:
    def albatross_alias(self):
        return 'node%d' % self.node_num
```

When the template is executed a unique name is generated for each checkbox. The exact HTML produced by the above fragment from the sample looks like this:

```
-<input type="checkbox" name="node12.selected" value="on">a
 |-<input type="checkbox" name="node2.selected" value="on">a
 | |-<input type="checkbox" name="node0.selected" value="on">a
 | \-<input type="checkbox" name="node1.selected" value="on">b
 \-<input type="checkbox" name="node11.selected" value="on">b
   |-<input type="checkbox" name="node6.selected" value="on">a
   | \-<input type="checkbox" name="node5.selected" value="on">a
   |   |-<input type="checkbox" name="node3.selected" value="on">a
   |   \-<input type="checkbox" name="node4.selected" value="on">b
   |-<input type="checkbox" name="node7.selected" value="on">b
   \-<input type="checkbox" name="node10.selected" value="on">c
     |-<input type="checkbox" name="node8.selected" value="on">a
     \-<input type="checkbox" name="node9.selected" value="on">b
```

The `alias` handling uses the fact that all Python objects are stored by reference. It obtains a reference to an existing object by resolving an expression and stores that reference under a new name. Since both the original expression and the new name are the same reference, the toolkit can modify the object referenced by the original expression by using the new name.

Looking further into the `samples/tree/tree1.py` code you will note that the tree being iterated is generated once and placed into the session. This ensures that the alias names generated always contain references to the nodes in the tree. If the tree was not entered into the session but was generated from scratch every request, the nodes referenced in the alias names would not be the same nodes as those in the tree so all input would be lost.

`checked` attribute

This attribute is generated in `radio` and `checkbox` input field types if the generated `value` (*value="..." attribute*) attribute matches the comparison value from `valueexpr` (*valueexpr="..." attribute*) (or the literal `'on'` for the `checkbox` input field type).

Refer to the documentation for individual input types for more details.

`expr="..."` attribute

For `text`, `password`, `submit`, `reset`, `hidden`, and `button` input field types the expression in the `expr` attribute is evaluated and the result is used to generate the `value` (*value="..." attribute*) attribute. If the result is `None` then no `value` attribute will be written.

For radio and checkbox input field types the expression in the expr attribute is evaluated and the result is compared with the generated value attribute to determine whether or not the checked (*checked attribute*) attribute should be written.

Refer to the documentation for individual input types for more details.

list attribute

If you are using an execution context that inherits from the NameRecorderMixin (nearly all do — see chapter *Prepackaged Application and Execution Context Classes*) then the execution context will raise a ApplicationError exception if multiple instances of some types of input tag with the same name are added to a form. The list attribute of the <al-input> tag is used indicate that multiple instances are intentional.

The presence of the list attribute on an <al-input> tag makes the request merging in the NameRecorderMixin class place any browser request values for the field into a list (field not present is represented by the empty list).

The attribute must not be used for input field types radio, submit, and image. The attribute is ignored for the file input field type.

name="..." attribute

When determining the generated name attribute the tag looks for a number of attributes. Any supplied name attribute will be ignored if any of the following attributes are present; prevpage, nextpage (*prevpage="..." and nextpage="..." attributes*), treefold, treeselect, treeellipsis (*treeselect="...", treefold="..." and treeellipsis="..." attributes*), alias (*alias="..." attribute*), or nameexpr (*nameexpr="..." attribute*).

All of the attributes that automatically generate names and are looked up in the above sequence. The first of those attributes found will be used to determine the name used in the tag.

nameexpr="..." attribute

This attribute is ignored if any of the following attributes are present; prevpage, nextpage (*prevpage="..." and nextpage="..." attributes*), treefold, treeselect, treeellipsis (*treeselect="...", treefold="..." and treeellipsis="..." attributes*), or alias (:ref :*tag-input-alias*).

The expression in the value of the nameexpr attribute is evaluated to determine the generated name (*name="..." attribute*) attribute.

One shortcoming of the alias attribute is that you can only perform input on object attributes. The nameexpr enables you to perform input on list elements.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.names = ['John', 'Terry', 'Eric']
>>> albatross.Template(ctx, '<magic>', '''
... <al-for iter="i" expr="range(len(names))">
...  <al-input nameexpr="'names[%d]' % i.value()" whitespace>
... </al-for>
... ''').to_html(ctx)
>>> ctx.flush_content()
<input name="names[0]" value="John" />
<input name="names[1]" value="Terry" />
<input name="names[2]" value="Eric" />
```

When the browser request is merged into the execution context the names elements of the names list will be replaced.

`node="..."` attribute

The `node` attribute is used in conjunction with the `treeselect`, `treefold` and `treeellipsis` (*treeselect="...", treefold="..." and treeellipsis="..." attributes*) attributes. It is ignored otherwise.

When this attribute is present the node identified by evaluating the expression in the attribute value will be used when generating the `name` attribute.

The `name` (*name="..." attribute*) attribute is generated as follows:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> class Node:
...     def __init__(self, ino):
...         self.ino = ino
...     def albatross_alias(self):
...         return 'ino%d' % self.ino
...
>>> ctx.locals.node = Node(81489)
>>> albatross.Template(ctx, '<magic>', '''
... <al-input type="image" treeselect="n" node="node" src="/icons/face.gif" border="0" whites
... ''').to_html(ctx)
>>> ctx.flush_content()
<input type="image" src="/icons/face.gif" border="0" name="treeselect,n,ino81489" />
```

Refer to the documentation for `treeselect`, `treefold` and `treeellipsis` (*treeselect="...", treefold="..." and treeellipsis="..." attributes*) attributes for more information.

`noescape` attribute

The `noescape` attribute can be used in conjunction with the `text`, `password`, `submit`, `reset`, `hidden`, and `button` input fields, it is ignored otherwise.

When this attribute is present the `value` (*value="..." attribute*) attribute will not be escaped.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.oops = '&<>"\''
>>> albatross.Template(ctx, '<magic>', '''
... <al-input type="text" name="oops" noescape whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<input type="text" name="oops" value="&<>"'" />
```

`prevpage="..."` and `nextpage="..."` attributes

The `prevpage` and `nextpage` attributes respectively select the previous and next pages of an `<al-for>` `ListIterator` (*ListIterator Objects*).

An attribute value must be supplied that specifies the name of the iterator.

The `name` (*name="..." attribute*) attribute is generated as follows:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-input type="image" nextpage="i" src="/icons/right.gif" border="0" whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<input type="image" src="/icons/right.gif" border="0" name="nextpage,i" />
```

When merging the browser request the `NamespaceMixin.set_value()` (:ref *:mixin-namespace*) method looks for field names that contain commas. These names are split into *operation*, *iterator*, and optional *value* then the `set_backdoor()` method of the identified iterator is invoked.

During request merging the above example will execute code equivalent to the following.

```
ctx.locals.i.set_backdoor('nextpage', 'nextpage,i')
```

`treeselect="..."`, `treefold="..."` and `treeellipsis="..."` attributes

The `treeselect`, `treefold`, and `treeellipsis` attributes respectively select, open/close, or expand the ellipsis of an `<al-tree>` node via a `LazyTreeIterator` (*LazyTreeIterator Objects*) or `EllipsisTreeIterator` (*EllipsisTreeIterator Objects*) iterator.

These attributes are ignored if any of the following attributes are present; `prevpage`, or `nextpage` (*prevpage="..." and nextpage="..." attributes*).

An attribute value must be supplied that specifies the name of the `LazyTreeIterator` iterator.

The `name` (*name="..." attribute*) attribute is generated as follows:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> class Node:
...     def __init__(self, ino):
...         self.ino = ino
...     def albatross_alias(self):
...         return 'ino%d' % self.ino
...
>>> ctx.locals.tree = Node(81489)
>>> albatross.Template(ctx, '<magic>', '''
... <al-tree expr="tree" iter="n">
... <al-input type="image" treeselect="n" src="/icons/face.gif" border="0" whitespace>
... </al-tree>
... ''').to_html(ctx)
>>> ctx.flush_content()
<input type="image" src="/icons/face.gif" border="0" name="treeselect,n,ino81489" />
```

When merging the browser request the `NamespaceMixin.set_value()` (:ref *:mixin-namespace*) method looks for field names that contain commas. These names are split into *operation*, *iterator*, and optional *value* then the `set_backdoor()` method of the identified iterator is invoked.

During request merging the above example will execute code equivalent to the following.

```
ctx.locals.n.set_backdoor('treeselect', 'ino81489', 'treeselect,n,ino81489')
```

The “ino81489” string is generated by calling the `albatross_alias()` method for the tree node.

If the `node`(*node="..." attribute*) attribute is not specified, the node referenced by the current value of the iterator will be used to determine the alias.

`value="..."` attribute

When determining the generated `value` attribute the tag looks for a number of attributes. Any supplied `value` attribute will be ignored if either `expr` (*expr="..." attribute*) or `valueexpr` (*valueexpr="..." attribute*) attributes (depending upon input type) present.

If no `expr`, `valueexpr`, or `value` attribute is present then the value identified by the generated `name` (*name="..." attribute*) attribute will be taken from the local namespace. If this value is `None` then no `value` attribute will be written. The name used to look into the local namespace is the result of evaluating all `name` related attributes.

For input field types `radio` and `checkbox` the `valueexpr` attribute if present takes priority over and specified `value` attribute.

Refer to the documentation for individual input types for more details.

`valueexpr="..."` attribute

This attribute is used to generate a `value` (*value="..." attribute*) attribute for `radio` and `checkbox` input field types. It is ignored for in all other cases.

Refer to the documentation for individual input types for more details.

`type="..."` attribute (text, password, submit, reset, hidden, button)

The tag determines the generated `name` (*name="..." attribute*) from the `name` related attributes.

To determine the generated `value` (*value="..." attribute*) attribute the tag first looks for an `expr` (*expr="..." attribute*) attribute, then a `value` attribute, and then if that fails it looks up the generated `name` in the execution context.

If the generated `name` contains a non-empty value it will be written as the `name` attribute. If the generated `value` is not `None` it will be escaped and written as the `value` attribute. Escaping values makes all `&`, `<`, `>`, and `"` characters safe.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def input_add(self, *args):
...         print args
...
>>> ctx = Ctx('.')
>>> ctx.locals.zero = 0
>>> ctx.locals.zerostr = '0'
>>> ctx.locals.width = 5
>>> ctx.locals.height = 7
>>> ctx.locals.secret = 42
>>> ctx.locals.other_secret = '<"&'
>>> albatross.Template(ctx, '<magic>', '''
... <al-input name="zero" whitespace>
... <al-input name="zerostr" whitespace>
... <al-input name="width" whitespace>
... <al-input name="area" expr="width * height" whitespace>
... <al-input type="password" name="passwd" whitespace>
... <al-input type="submit" name="login" value="Login" whitespace>
... <al-input type="hidden" name="secret" whitespace>
... <al-input type="hidden" name="other_secret" whitespace>
... ''').to_html(ctx)
('text', 'zero', 0, False)
('text', 'zerostr', '0', False)
('text', 'width', 5, False)
('text', 'area', 35, False)
('password', 'passwd', None, False)
('submit', 'login', 'Login', False)
('hidden', 'secret', 42, False)
('hidden', 'other_secret', '<"&', False)
>>> ctx.flush_content()
<input name="zero" value="0" />
<input name="zerostr" value="0" />
<input name="width" value="5" />
<input name="area" value="35" />
<input type="password" name="passwd" />
```

```
<input type="submit" name="login" value="Login" />
<input type="hidden" name="secret" value="42" />
<input type="hidden" name="other_secret" value="&lt;&quot;&amp;" />
```

After writing all tag attributes the execution context `input_add()` method is called with the following arguments; input field type (`'text'`, `'password`, `'submit`, `'reset`, `'hidden`, or `'button'`), the generated `name`, the generated `value`, and a flag indicating whether or not the `list` (*list attribute*) attribute was present.

Application code handling browser requests typically determines the `submit` input pressed by the user via the execution context `req_equals()` method. The `req_equals()` method simply tests that the named input is present in the browser request and contains a non-empty value.

For example:

```
def page_process(ctx):
    if ctx.req_equals('login'):
        user = process_login(ctx.locals.username, ctx.locals.passwd)
        if user:
            ctx.locals._user = user
            ctx.add_session_vars('_user')
            ctx.set_page('home')
```

`type="..."` attribute (radio)

The tag determines the generated `name` (*name="..." attribute*) from the `name` related attributes. Then an internal comparison value is determined by evaluating the `expr` (*expr="..." attribute*) attribute if it is present, or by looking up the generated `name` in the execution context.

If the comparison value equals the generated `value` (*value="..." attribute*) attribute then the `checked` (*checked attribute*) attribute is written. Both values are converted to string before being compared.

To determine the generated `value` attribute the tag first looks for a `valueexpr` (*valueexpr="..." attribute*) attribute, then a `value` attribute.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def input_add(self, *args):
...         print args
...
>>> ctx = Ctx('.')
>>> ctx.locals.swallow = 'African'
>>> albatross.Template(ctx, '<magic>', '''
... <al-input type="radio" name="swallow" value="African" whitespace>
... <al-input type="radio" name="swallow" value="European" whitespace>
... ''').to_html(ctx)
('radio', 'swallow', 'African', False)
('radio', 'swallow', 'European', False)
>>> ctx.flush_content()
<input type="radio" name="swallow" value="African" checked />
<input type="radio" name="swallow" value="European" />
```

The `expr` attribute can be used to generate the internal comparison value. This is then compared with the `value` attribute to control the state of the `checked` attribute.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def input_add(self, *args):
```

```
...         print args
...
>>> ctx = Ctx('.')
>>> ctx.locals.swallows = ['African', 'European']
>>> ctx.locals.num = 0
>>> albatross.Template(ctx, '<magic>', '''
... <al-input type="radio" name="swallow" expr="swallows[num]" value="African" whitespace>
... <al-input type="radio" name="swallow" expr="swallows[num]" value="European" whitespace>
... ''').to_html(ctx)
('radio', 'swallow', 'African', False)
('radio', 'swallow', 'European', False)
>>> ctx.flush_content()
<input type="radio" name="swallow" value="African" checked />
<input type="radio" name="swallow" value="European" />
```

The `valueexpr` attribute can be used to dynamically generate the `value` attribute.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def input_add(self, *args):
...         print args
...
>>> ctx = Ctx('.')
>>> ctx.locals.swallows = ['African', 'European']
>>> ctx.locals.num = 0
>>> albatross.Template(ctx, '<magic>', '''
... <al-for iter="s" expr="swallows">
... <al-input type="radio" name="swallow" expr="swallows[num]" valueexpr="s.value()" whitespa
... </al-for>
... ''').to_html(ctx)
('radio', 'swallow', 'African', False)
('radio', 'swallow', 'European', False)
>>> ctx.flush_content()
<input type="radio" name="swallow" value="African" checked />
<input type="radio" name="swallow" value="European" />
```

After writing all tag attributes the execution context `input_add()` method is called with the arguments; input field type (`'radio'`), the generated `name`, the generated `value`, and a flag indicating whether or not the `list` (*list attribute*) attribute was present.

`type="..."` attribute (checkbox)

The tag determines the generated `name` (*name="..." attribute*) from the `name` related attributes. Then an internal comparison value is determined by evaluating the `expr` (*expr="..." attribute*) attribute if it is present, or by looking up the generated `name` in the execution context.

If the comparison value equals the generated `value` (*value="..." attribute*) attribute then the `checked` (*checked attribute*) attribute is written. Both values are converted to string before being compared.

If the internal comparison value is either a list or tuple the `checked` attribute is written if the generated `value` attribute is present in the list/tuple.

To determine the generated `value` attribute the tag first looks for a `valueexpr` (*valueexpr="..." attribute*) attribute, then a `value` attribute, and then if that fails it defaults to the value `'on'`.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def input_add(self, *args):
```

```
...         print args
...
>>> ctx = Ctx('.')
>>> ctx.locals.menu = ['spam', 'eggs']
>>> ctx.locals.eric = 'half'
>>> ctx.locals.parrot = 'on'
>>> ctx.locals.halibut = 'off'
>>> albatross.Template(ctx, '<magic>', '''
... <al-input type="checkbox" name="menu" value="spam" whitespace>
... <al-input type="checkbox" name="menu" value="eggs" whitespace>
... <al-input type="checkbox" name="menu" value="bacon" whitespace>
... <al-input type="checkbox" name="eric" value="half" whitespace>
... <al-input type="checkbox" name="parrot" whitespace>
... <al-input type="checkbox" name="halibut" whitespace>
... ''').to_html(ctx)
('checkbox', 'menu', 'spam', False)
('checkbox', 'menu', 'eggs', False)
('checkbox', 'menu', 'bacon', False)
('checkbox', 'eric', 'half', False)
('checkbox', 'parrot', 'on', False)
('checkbox', 'halibut', 'on', False)
>>> ctx.flush_content()
<input type="checkbox" name="menu" value="spam" checked />
<input type="checkbox" name="menu" value="eggs" checked />
<input type="checkbox" name="menu" value="bacon" />
<input type="checkbox" name="eric" value="half" checked />
<input type="checkbox" name="parrot" value="on" checked />
<input type="checkbox" name="halibut" value="on" />
```

After writing all tag attributes the execution context `input_add()` method is called with the arguments; input field type (`'checkbox'`), the generated `name`, the generated `value`, and a flag indicating whether or not the `list` (*list attribute*) attribute was present.

`type="..."` attribute (image)

The tag determines the generated `name` (*name="..." attribute*) from the `name` related attributes.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def input_add(self, *args):
...         print args
...
>>> ctx = Ctx('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-input type="image" nextpage="m" src="/icons/right.gif" whitespace>
... ''').to_html(ctx)
('image', 'nextpage,m', None, False)
>>> ctx.flush_content()
<input type="image" src="/icons/right.gif" name="nextpage,m" />
```

After writing all tag attributes the execution context `input_add()` method is called with the arguments; input field type (`'image'`), the generated `name`, `None`, and a flag indicating whether or not the `list` (:ref :*tag-input-list*) attribute was present.

When a browser submits input to an `image` input it sends an `x` and `y` value for the field. These are saved as attributes of the field.

For example, if an image input named `map` was clicked by the user, then the code to detect and process the input would look something like this:

```
def page_process(ctx):
    if ctx.req_equals('map'):
        map_clicked_at(ctx.locals.map.x, ctx.locals.map.y)
```

`type="..."` attribute (file)

The tag determines the generated `name` (*name="..." attribute*) from the `name` related attributes.

If you are using an execution context that inherits from the `NameRecorderMixin` (*RecorderMixin Classes*) then using this input field type will automatically cause the enclosing `<al-form>` (*<al-form>*) tag to include an `enctype="multipart/form-data"` (*enctype="..." attribute*) attribute.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def input_add(self, *args):
...         print args
...
>>> ctx = Ctx('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-input type="file" name="resume" whitespace>
... ''').to_html(ctx)
('file', 'resume', None, False)
>>> ctx.flush_content()
<input type="file" name="resume" />
```

After writing all tag attributes the execution context `input_add()` method is called with the arguments; input field type (`'file'`), the generated `name`, `None`, and a flag indicating whether or not the `list` (:ref :*tag-input-list*) attribute was present.

The request merging allows the user to submit more than one file in a `file` input field. To simplify application code the `Request` always returns a list of `FileField` objects for `file` inputs.

Application code to process `file` inputs typically looks like the following:

```
def page_process(ctx):
    if ctx.req_equals('resume'):
        for r in ctx.locals.resume:
            if r.filename:
                save_uploaded_resume(r.filename, r.file.read())
```

8.2.3 `<al-select>`

Albatross browser request merging depends upon the functionality provided by the `<al-select>` tag. If you do no use this tag in applications then the standard request merging will not work.

To determine the `name` (*name="..." attribute*) attribute in the generated tag a number of attributes are used. The generated name is evaluated in the execution context to determine an internal compare value.

The compare value is used to control which option tags are generated with the `selected` (*selected attribute*) attribute. `<select>` tags in multi- select mode are supported by list or tuple compare values.

The `<al-select>` tag can automatically generate the list of enclosed `<option>` tags using the `optionexpr` (*optionexpr="..." attribute*) attribute, or can work with enclosed `<al-option>` (*<al-option>*) tags.

`alias="..."` attribute

The value of the `alias` attribute is passed to the `make_alias()` method of the execution context. The return value is then used as the generated `name` (*name="..." attribute*) attribute.

The execution context `make_alias()` method splits the `alias` attribute at the last '.' and resolves the left hand side to an object reference. The `albatross_alias()` method is then called on that object and the result is combined with the '.' and the right hand side of the of the `alias` attribute to produce the generated `name` attribute. The resolved object is entered in the the local namespace and the session using the name returned by the `albatross_alias()` method.

Refer to the documentation of the `alias` attribute of the `<al-input>` tag (*alias="..." attribute*) for an example of the mechanism described above.

`list` attribute

If you are using an execution context that inherits from the `NameRecorderMixin` (nearly all do — see chapter *Prepackaged Application and Execution Context Classes*) then the execution context will raise a `ApplicationError` exception if multiple instances of a non-multi-select `<al-select>` tag with the same name are added to a form. The `list` attribute is used indicate that multiple instances are intentional.

The presence of the `list` attribute on an `<al-select>` tag makes the request merging in the `NameRecorderMixin` class place any browser request values for the field into a list (field not present is represented by the empty list).

`multiple` attribute

For the purposes of the `NameRecorderMixin` class, this attribute performs the same role as the `list` (*list attribute*) attribute. It tells the browser request merging to place all input values into a list (field not present is represented by the empty list).

`name="..."` attribute

When determining the generated `name` attribute the tag looks for a number of attributes. Any supplied `name` attribute will be ignored if either the `alias` (*alias="..." attribute*) or `nameexpr` (*nameexpr="..." attribute*) attributes are present.

`nameexpr="..."` attribute

This attribute is ignored if the `alias` (*alias="..." attribute*) attribute is present.

The expression in the value of the `nameexpr` attribute is evaluated to determine the generated `name` (*name="..." attribute*) attribute.

One shortcoming of the `alias` attribute is that you can only perform input on object attributes. The `nameexpr` enables you to perform input on list elements.

Refer to the documentation of the `nameexpr` attribute of the `<al-input>` tag (*nameexpr="..." attribute*) for an example.

`noescape` attribute

The `noescape` attribute is used with the `optionexpr` (*optionexpr="..." attribute*) attribute to suppress escaping of each option value returned by the expression.

`optionexpr="..."` attribute

If this attribute is present the expression in the attribute value is evaluated to determine a sequence of option values. One `<option>` tag is generated for each item in the sequence.

When this attribute is not present all of the directly enclosed `<al-option>` (*<al-option>*) tags are processed to generate the enclosed `<option>` tags.

If an item in the `optionexpr` sequence is not a tuple, it is converted to string and then compared with the comparison value derived from the `name` (*name="..." attribute*) attribute.

To support multiple selected `<option>` tags the comparison value must be either a list or tuple.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.sel1 = 3
>>> ctx.locals.sel2 = (2,3)
>>> albatross.Template(ctx, '<magic>', '''
... <al-select name="sel1" optionexpr="range(5)" whitespace/>
... <al-select name="sel2" optionexpr="range(5)" multiple whitespace/>
... ''').to_html(ctx)
>>> ctx.flush_content()
<select name="sel1"><option>0</option>
<option>1</option>
<option>2</option>
<option selected>3</option>
<option>4</option>
</select>
<select multiple name="sel2"><option>0</option>
<option>1</option>
<option selected>2</option>
<option selected>3</option>
<option>4</option>
</select>
```

If an item in the `optionexpr` sequence is a tuple it must contain two values. The first value is used to specify the `value` attribute of the generated `<option>` tag and the second value provides the `<option>` tag content.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.menu = [(1, 'Spam'), (2, 'Eggs'), (3, 'Bacon')]
>>> ctx.locals.sel = 1
>>> albatross.Template(ctx, '<magic>', '''
... <al-select name="sel" optionexpr="menu" whitespace/>
... ''').to_html(ctx)
>>> ctx.flush_content()
<select name="sel"><option selected value="1">Spam</option>
<option value="2">Eggs</option>
<option value="3">Bacon</option>
</select>
```

All values generated by the `optionexpr` method are escaped to make all `&`, `<`, `>`, and `"` characters safe.

8.2.4 `<al-option>`

Unless explicitly overridden, the `selected` attribute is controlled by the comparison of the value of the enclosing `<al-select>` (*<al-select>*) tag with the evaluated value of the `<al-option>` tag.

The value of the `<al-option>` tag is specified either by evaluating the `valueexpr` (*valueexpr="..." attribute*) attribute, or the `value` (:ref :*tag-option-value*) attribute, or if neither attribute is present, by the content enclosed by the `<al-option>` tag. The enclosed content of the tag is evaluated before it is compared. This allows the content to be generated using other Albatross tags.

Albatross browser request merging depends upon the functionality provided by the `<al-option>` tag. If you do no use this tag in applications then the standard request merging will not work.

For example — this shows how the `<al-option>` content is evaluated before it is compared with the `<al-select>` value:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.opt = 'spam'
>>> ctx.locals.sel = 'spam'
>>> albatross.Template(ctx, '<magic>', '''
... <al-select name="sel">
...  <al-option><al-value expr="opt"></al-option>
...  <al-option>eggs</al-option>
... </al-select whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<select name="sel"><option selected>spam</option><option>eggs</option></select>
```

`selected` attribute

The `selected` attribute overrides the value comparison logic. When the `selectedbool` form is used, this allows the `selected` flag to be controlled via arbitrary logic.

`value="..."` attribute

Use the `value` attribute to specify a value to be compared with the comparison value of the enclosing `<al-select>` (*<al-select>*) tag.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.opt = 3
>>> ctx.locals.sel = ['spam', 'eggs']
>>> albatross.Template(ctx, '<magic>', '''
... <al-select name="sel" multiple>
...  <al-option value="spam">Spam <al-value expr="opt"> times
...  </al-option>
...  <al-option>eggs</al-option>
...  <al-option>bacon</al-option>
... </al-select whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<select multiple name="sel"><option value="spam" selected>Spam 3 times
 </option><option selected>eggs</option><option>bacon</option></select>
```

`valueexpr="..."` attribute

Use the `valueexpr` attribute to specify an expression to be evaluated to derive the value to be compared with the comparison value of the enclosing `<al-select>` (*<al-select>*) tag.

If the `valueexpr` attribute evaluates to a 2-tuple, the first item becomes the value and the second becomes the label.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.spam = 'manufactured meat'
>>> ctx.locals.potato = ('mash', 'Mashed Potato')
>>> ctx.locals.sel = ['manufactured meat', 'eggs']
>>> albatross.Template(ctx, '<magic>', '''
... <al-select name="sel" multiple>
...  <al-option valueexpr="spam" />
...  <al-option valueexpr="potato" />
...  <al-option>eggs</al-option>
...  <al-option>bacon</al-option>
... </al-select whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<select multiple name="sel"><option selected>manufactured meat</option><option value="mash">M
```

`label="..."` attribute

Use the `label` attribute to specify the control label. This overrides the body of the `<al-option>` tag.

`labelexpr="..."` attribute

Use the `labelexpr` attribute to specify an expression to be evaluated to derive the control label. This overrides the body of the `<al-option>` tag.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.spam = 'manufactured meat'
>>> ctx.locals.sel = ['spam', 'eggs']
>>> albatross.Template(ctx, '<magic>', '''
... <al-select name="sel" multiple>
...  <al-option labelexpr="spam">spam</al-option>
...  <al-option>eggs</al-option>
... </al-select whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<select multiple name="sel"><option value="spam" selected>manufactured meat</option><option s
```

8.2.5 `<al-textarea>`

Albatross browser request merging depends upon the functionality provided by the `<al-textarea>` tag. If you do no use this tag in applications then the standard request merging will not work.

`alias="..."` attribute

The value of the `alias` attribute is passed to the `make_alias()` method of the execution context. The return value is then used as the generated `name` (*name="..." attribute*) attribute.

The execution context `make_alias()` method splits the `alias` attribute at the last '.' and resolves the left hand side to an object reference. The `albatross_alias()` method is then called on that object and the result is combined with the '.' and the right hand side of the of the `alias` attribute to produce the generated `name` attribute. The resolved object is entered in the the local namespace and the session using the name returned by the `albatross_alias()` method.

Refer to the documentation of the `alias` attribute of the `<al-input>` tag (*alias="..." attribute*) for an example of the mechanism described above.

`list` attribute

If you are using an execution context that inherits from the `NameRecorderMixin` (nearly all do — see chapter *Prepackaged Application and Execution Context Classes*) then the execution context will raise a `ApplicationError` exception if multiple instances of an `<al-textarea>` tag with the same name are added to a form. The `list` attribute is used indicate that multiple instances are intentional.

The presence of the `list` attribute on an `<al-textarea>` tag makes the request merging in the `NameRecorderMixin` class place any browser request values for the field into a list (field not present is represented by the empty list).

`name="..."` attribute

When determining the generated `name` attribute the tag looks for a number of attributes. Any supplied `name` attribute will be ignored if either the `alias` (*alias="..." attribute*) or `nameexpr` (*nameexpr="..." attribute*) attributes are present.

If the value identified by the generated `name` attribute does not exist in the execution context then the enclosed content will be supplied as the initial tag value.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> text = '''
... <al-textarea name="msg">
...  Type in some text
... </al-textarea whitespace>
... '''
>>> albatross.Template(ctx, '<magic>', text).to_html(ctx)
>>> ctx.flush_content()
<textarea name="msg">Type in some text
</textarea>
>>> ctx.locals.msg = 'This came from the program'
>>> albatross.Template(ctx, '<magic>', text).to_html(ctx)
>>> ctx.flush_content()
<textarea name="msg">This came from the program</textarea>
```

Before the tag value is written it is escaped to make all `&`, <, >, and `"` characters safe.

`nameexpr="..."` attribute

This attribute is ignored if the `alias` (*alias="..." attribute*) attribute is present.

The expression in the value of the `nameexpr` attribute is evaluated to determine the generated `name` (*name="..." attribute*) attribute.

One shortcoming of the `alias` attribute is that you can only perform input on object attributes. The `nameexpr` enables you to perform input on list elements.

Refer to the documentation of the `nameexpr` attribute of the `<al-input>` tag (*nameexpr="..." attribute*) for an example.

`noescape` attribute

The `noescape` attribute is used to suppress escaping of the execution context value associated with the `name` (*name="..." attribute*) attribute.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.msg = 'Should escape < & >...'
>>> albatross.Template(ctx, '<magic>', '''
... <al-textarea name="msg" noescape whitespace/>
... ''').to_html(ctx)
>>> ctx.flush_content()
<textarea name="msg">Should escape < & >...</textarea>
```

8.2.6 `<al-a>`

This tag acts as an enhanced version of the standard HTML `<a>` tag.

`expr="..."` attributes

This attribute is ignored is any of the following attributes are present; `prevpage`, `nextpage` (*prevpage="..." and nextpage="..." attributes*), `treefold`, `treeselect`, or `treeellipsis` (*treeselect="...", treefold="..." and treeellipsis="..." attributes*).

The specified expression is evaluated to generate an `href` (*href="..." attributes*) attribute. The generated attribute is then processed as per the `href` attribute.

`href="..."` attributes

This attribute is ignored is any of the following attributes are present; `prevpage`, `nextpage` (*prevpage="..." and nextpage="..." attributes*), `treefold`, `treeselect`, `treeellipsis` (*treeselect="...", treefold="..." and treeellipsis="..." attributes*), or `expr` (*expr="..." attributes*).

When the `expr` attribute is used, then generated value is processed in the same as a value supplied in the `href` attribute.

If the `href` does not contain a '?' (separates the path from the query), but does contain a '=' then the `href` is rewritten as *current_url*?*href*.

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def current_url(self):
...         return 'magic'
...
>>> ctx = Ctx('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-a href="login=1">Login</al-a whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<a href="magic?login=1">Login</a>
```

If the `href` does not contain either a '?' or a '=' then the `href` is assumed to be a page identifier so it is transformed into a redirect url by the `redirect_url()` execution context method.

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def current_url(self):
...         return 'magic'
...     def redirect_url(self, loc):
...         return 'here/%s' % loc
...
>>> ctx = Ctx('.')
>>> ctx.locals.name = 'eric'
>>> albatross.Template(ctx, '<magic>', '''
... <al-a expr="'login=%s' % name">Login</al-a whitespace>
... <al-a expr="'remote?login=%s' % name">Login</al-a whitespace>
... <al-a href="page">Login</al-a whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<a href="magic?login=eric">Login</a>
<a href="remote?login=eric">Login</a>
<a href="here/page">Login</a>
```

`node="..."` attribute

The `node` attribute is used in conjunction with the `treeselect`, `treefold` and `treeellipsis` (*treeselect="...", treefold="..." and treeellipsis="..." attributes*) attributes. It is ignored otherwise.

When this attribute is present the node identified by evaluating the expression in the attribute value will be used when generating the `href` (*href="..." attributes*) attribute.

`prevpage="..."` and `nextpage="..."` attributes

The `prevpage` and `nextpage` attributes generate an `href` (*href="..." attributes*) attribute that respectively selects the previous and next pages of an `<al-for>` `ListIterator` (*ListIterator Objects*).

The attribute value specifies the name of the iterator.

The generated `href` attribute is of the form *current_url*?*name*,*iter*=1 where *current_url* is the path component returned from the Python `urlparse.urlparse()` function (via the execution context `current_url()` method), *name* is either `prevpage` or `nextpage`, and *iter* is the specified iterator.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext):
...     def current_url(self):
...         return 'magic'
...
>>> ctx = Ctx('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-a nextpage="m">Next Page</al-a whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<a href="magic?nextpage,m=1">Next Page</a>
```

`treeselect="..."`, `treefold="..."` and `treeellipsis="..."` attributes

These attributes are ignored if any either the `prevpage` (*prevpage="..." and nextpage="..." attributes*), or `nextpage` attributes are present.

The `treeselect`, `treefold`, and `treeellipsis` attributes generate an `href` (*href="..." attributes*) attribute that respectively select, open/close, or expand the ellipsis of an `<al-tree>` (*<al-tree>*) node via a `LazyTreeIterator` (*LazyTreeIterator Objects*) or `EllipsisTreeIterator` (*EllipsisTreeIterator Objects*) iterator.

Refer to the `<al-input>` tag for more information on how to use these attributes (*treeselect="...", treefold="..." and treeellipsis="..." attributes*).

If the `node` (*node="..." attribute*) attribute if it is present it defines the node to operate upon. Otherwise the node operated upon is the current value of the `LazyTreeIterator` iterator.

The attribute value specifies the name of the `LazyTreeIterator` iterator.

The generated `href` attribute is of the form *current_url*?*name*,*iter*,*alias*=1 where *current_url* is the path component returned from the Python `urlparse.urlparse()` function (via the execution context `current_url()` method), *name* is either `treeselect`, `treefold` or `treeellipsis`, *iter* is the specified iterator, and *alias* is the values returned by the `albatross_alias()` method of the specified node.

8.2.7 `<al-img>`

Use this tag to dynamically generate the `src` (*src="..." attribute*) attribute of an `<img>` tag.

`expr="..."` attribute

You must supply an `expr` attribute containing an expression that is evaluated to generate the output `src` (*src="..." attribute*) attribute of the `<img>` tag.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.src = 'http://icons.r.us/spam.png'
>>> albatross.Template(ctx, '<magic>', '''
... <al-img expr="src" whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<img src="http://icons.r.us/spam.png" />
```

`noescape` attribute

The `noescape` attribute can be used to suppress escaping of the `src` (:ref :*tag-img-src*) attribute.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.url = 'http://www.com/img?a=1&b=2'
>>> albatross.Template(ctx, '<magic>', '''
... <al-img expr="url" noescape whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
<img src="http://www.com/img?a=1&b=2" />
```

`src="..."` attribute

This attribute is ignored. Use the `expr` (*expr="..." attribute*) attribute to generate the `<src>` attribute.

8.3 Other HTML Tags

Arbitrary HTML tags can access the templating engine by prefixing the tag with "al-". Attributes of the tag can then be selectively evaluated to derive their value:

- Appending "expr" to the attribute name causes the value of the attribute to be evaluted and the results of the evaluation substituted for the value.
- Appending "bool" results in the attribute value being evaluated in a boolean context. If the result is `True`, a boolean HTML attribute is emitted, and if the result is `False`, no attribute is emitted.
- Any other attributes are passed through unchanged.

For example:

```
<al-td colspanexpr="i.span()">
```

could produce

```
<td colspan="3">
```

In the following example:

```
<al-input name="abc.value" disabledbool="abc.isdisabled()">
```

If `abc.isdisabled()` evaluates as `True`, then the `disabled` attribute is emitted:

```
<input name="abc.value" disabled>
```

But if `abc.isdisabled()` evaluates as `False`, then the `disabled` attribute is suppressed entirely:

```
<input name="abc.value">
```

In the following example, we change the styling of alternate rows in a table:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.names = ['Alex', 'Fred', 'Guido', 'Martin', 'Raymond', 'Tim']
>>> albatross.Template(ctx, '<magic>', '''
... <table>
...  <al-for iter="i" expr="names">
...   <al-tr classexpr="['light', 'dark'][i.index() % 2]" whitespace>
...    <td><al-value expr="i.value()" /></td>
...   </al-tr>
...  </al-for>
... </table>
... ''').to_html(ctx)
>>> ctx.flush_content()
<table>
 <tr class="light">
   <td>Alex</td>
  </tr><tr class="dark">
   <td>Fred</td>
  </tr><tr class="light">
   <td>Guido</td>
  </tr><tr class="dark">
   <td>Martin</td>
  </tr><tr class="light">
   <td>Raymond</td>
  </tr><tr class="dark">
   <td>Tim</td>
  </tr></table>
```

8.4 Execution and Control Flow

Tags in this section provide just enough programming capability to allow template files to react to and format values from the execution context.

8.4.1 `<al-require>`

This tag is used to specify the minimum version of the Albatross templating system that will correctly parse your template, or to specify templating features (that may be implemented by extension modules) that are required to parse your template.

If the templating system has a lower version number, or the extension feature is not available, an `ApplicationError` Exception is raised when the template is parsed.

`version="..."` attribute

This attribute specifies the minimum version of the Albatross templating system required to correctly parse your template. Specify the lowest version that will correctly parse your template.

Templating Version	Albatross Version	Template feature
1	up to 1.20	
2	1.30 and up	prefixing any tag with `al-` now allows any attribute to be evaluated

`feature="..."` attribute

If the `feature` attribute is present, it specifies a comma-separated list of templating features that will be required to correctly parse your template. At present, no features are available.

8.4.2 `<al-include>`

Use this tag to load and execute another template file at the current location. You can specify the name of the included template file by name using the `name` (*name="..." attribute*) attribute or by expression using the `expr` (*expr="..." attribute*) attribute.

`expr="..."` attribute

If the `expr` attribute is present it is evaluated when the template is executed to generate the name of a template file. The specified template file is loaded and executed with the output replacing the `<al-include>` tag.

For example:

```
>>> open('other.html', 'w').write('name = "<al-value expr="name">"')
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.name = 'other.html'
>>> albatross.Template(ctx, '<magic>', '''
... Inserting <al-value expr="name">: <al-include expr="name"> here.
... ''').to_html(ctx)
>>> ctx.flush_content()
Inserting other.html: name = "other.html" here.
```

`name="..."` attribute

This attribute is ignored if the `expr` attribute is present.

When the template is executed the specified template file is loaded and executed with the output replacing the `<al-include>` tag.

For example:

```
>>> open('other.html', 'w').write('name = "<al-value expr="name">"')
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.name = 'other.html'
>>> albatross.Template(ctx, '<magic>', '''
... Inserting other.html: <al-include name="other.html"> here.
... ''').to_html(ctx)
>>> ctx.flush_content()
Inserting other.html: name = "other.html" here.
```

8.4.3 `<al-comment>`

This tag suppresses the execution and output of any contained content, although the contained content must be syntactically correct.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... Before,<al-comment>
...  This will not be executed
... </al-comment> after.
... ''').to_html(ctx)
>>> ctx.flush_content()
Before, after.
```

8.4.4 `<al-flush>`

When the template file interpreter encounters an `<al-flush>` during execution it flushes all accumulated HTML to output.

Usually HTML is accumulated in the execution context and is not sent to the output until the `flush_content()` is called. This gives programs the opportunity to handle exceptions encountered during template execution without partial output leaking to the browser.

When the program is performing an operation that runs for some time this behaviour may give user the impression that the application has entered an infinite loop. In these cases it is usually a good idea to provide incremental feedback to the user by placing `<al-flush>` tags in your template files.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... Processing results...
... <al-flush>
... Finished
... ''').to_html(ctx)
Processing results...
>>> ctx.flush_content()
Finished
```

8.4.5 `<al-if>`/`<al-elif>`/`<al-else>`

Use of these tags parallels the `if`/`elif`/`else` keywords in Python.

The `<al-if>` tag is a content enclosing tag while `<al-elif>` and `<al-else>` are empty tags that partition the content of the enclosing `<al-if>` tag.

`expr="..."` attribute

The `expr` attribute is used in the `<al-if>` and `<al-elif>` tags to specify a test expression. The expression is evaluated when the template is executed.

If the text expression in the `expr` attribute of the `<al-if>` tag evaluates to a `TRUE` value then the enclosed content up to either the next `<al-elif>` or `<al-else>` tag will be executed.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.a = 10
>>> albatross.Template(ctx, '<magic>', '''
... <al-if expr="a < 15">
...  a (<al-value expr="a">) is less than 15.
... <al-else>
...  a (<al-value expr="a">) is greater than or equal to 15.
... </al-if>
... ''').to_html(ctx)
>>> ctx.flush_content()
a (10) is less than 15.
```

If the expression in the `expr` attribute of the `<al-if>` tag evaluates `FALSE` then the enclosed content following the `<al-else>` tag is executed.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.a = 20
>>> albatross.Template(ctx, '<magic>', '''
... <al-if expr="a < 15">
...  a (<al-value expr="a">) is less than 15.
... <al-else>
...  a (<al-value expr="a">) is greater than or equal to 15.
... </al-if>
... ''').to_html(ctx)
>>> ctx.flush_content()
a (20) is greater than or equal to 15.
```

The `<al-elif>` tag is used to chain a number of expression that are tested in sequence. The first test that evaluates `TRUE` determines the content that is executed.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.a = 30
>>> albatross.Template(ctx, '<magic>', '''
... <al-if expr="a < 15">
...  a (<al-value expr="a">) is less than 15.
... <al-elif expr="a < 25">
...  a (<al-value expr="a">) is greater than or equal to 15 and less than 25.
... <al-elif expr="a < 35">
...  a (<al-value expr="a">) is greater than or equal to 25 and less than 35.
... <al-else>
...  a (<al-value expr="a">) is greater than or equal to 25.
... </al-if>
... ''').to_html(ctx)
>>> ctx.flush_content()
a (30) is greater than or equal to 25 and less than 35.
```

8.4.6 `<al-value>`

This tag allows you to evaluate simple expressions and write the result to output.

`date="..."` attribute

If a `date` attribute is specified then the enclosed format string is passed to the Python `time.strftime()` function along with the result of the expression in the `expr` (*expr="..." attribute*) attribute. The result of `time.strftime()` is then written to the output.

For example:

```
>>> import albatross
>>> import time
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... The time is <al-value expr="time.mktime((2001,12,25,1,23,45,0,0,-1))"
...                       date="%H:%M:%S" whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
The time is 01:23:45
```

`expr="..."` attribute

This attribute must the specified. It contains an expression that is evaluated when the template is executed and the result is written as a string to the output.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.items = ['pencil', 'eraser', 'lunchbox']
>>> albatross.Template(ctx, '<magic>', '''
... There are <al-value expr="len(items)" whitespace> items
... ''').to_html(ctx)
>>> ctx.flush_content()
There are 3 items
```

`lookup="..."` attribute

When the `lookup` attribute is specified the result of the expression in the `expr` (*expr="..." attribute*) attribute is used to retrieve content from the lookup table named in the `lookup` attribute. This is a very useful way to separate the internal representation of program value from the presentation of that value.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.simple = 0
>>> ctx.locals.works = 1
>>> albatross.Template(ctx, '<magic>', '''
... <al-lookup name="bool"><al-item expr="0">FALSE</al-item>TRUE</al-lookup>
... Simple: <al-value expr="simple" lookup="bool" whitespace>
...  Works: <al-value expr="works" lookup="bool" whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
Simple: FALSE
 Works: TRUE
```

Please refer to the `<al-lookup>` tag reference for an explanation of that tag and more complex examples.

`noescape` attribute

If the `noescape` attribute is present then the value is not escaped. Only use this attribute when you are sure that the result of the expression is safe. Without this attribute all `&`, `<`, `>`, and `"` are escaped.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.field = '<img src="http://rude.pictures.r.us/">'
>>> albatross.Template(ctx, '<magic>', '''
... Safe: <al-value expr="field" whitespace>
... Oops: <al-value expr="field" noescape whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
Safe: &lt;img src=&quot;http://rude.pictures.r.us/&quot;&gt;
Oops: <img src="http://rude.pictures.r.us/">
```

8.4.7 `<al-exec>`

This tag allows you to place arbitrary Python code in a template file.

`expr="..."` attribute

The expression is specified in the `expr` attribute. It is compiled using *kind* = `'exec'` and evaluated when the template is executed.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-exec expr="
... results = []
... for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            results.append('%d equals %d * %d' % (n, x, n/x))
...            break
...     else:
...          results.append('%d is a prime number' % n)
... ">
... <al-for iter="l" expr="results">
...  <al-value expr="l.value()" whitespace>
... </al-for>
... ''').to_html(ctx)
>>> ctx.flush_content()
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

If you need to include the same quote character used to enclose the attribute value in your expression you can escape it using a backslash ("$\").

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', r'''
... <al-exec expr="a = '\"'">
... a = <al-value expr="a" whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
a = &quot;
```

8.4.8 `<al-for>`

This tag implements a loop in almost the same way as the `for` keyword in Python.

The tag uses an instance of the `ListIterator` (*ListIterator Objects*) identified in the local namespace by the `iter` (*iter="..." attribute*) attribute to iterate over the sequence defined by the expression in the `expr` (*expr="..." attribute*) attribute.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-for iter="i" expr="range(15)" whitespace="indent">
...  <al-value expr="i.value()">
... </al-for whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Note that you must use the `value()` method of the iterator to retrieve the current sequence value, or set a template namespace name via the `vars` attribute into which it will be stored.

When using pagination mode via the `pagesize` (*pagesize="..." attribute*) attribute the `prevpage` and `nextpage` attributes of the `<al-input>` (*<al-input>*) and `<al-a>` (*<al-a>*) tags can be used to automatically page forwards and backwards through a sequence.

The following simulates pagination via the `set_backdoor()` `ListIterator` method and shows other data that is maintained by the iterator.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.seq = range(9)
>>> t = albatross.Template(ctx, '<magic>', '''
... pagesize has_prevpage has_nextpage index start count value
... ----------------------------------------------------------
... <al-for iter="i" expr="seq" pagesize="3">
... <al-value expr="'%8s' % i.pagesize()">
... <al-value expr="'%13s' % i.has_prevpage()">
... <al-value expr="'%13s' % i.has_nextpage()">
... <al-value expr="'%6s' % i.index()">
... <al-value expr="'%6s' % i.start()">
... <al-value expr="'%6s' % i.count()">
... <al-value expr="'%6s' % i.value()" whitespace>
... </al-for>''')
>>> t.to_html(ctx)
>>> ctx.locals.i.set_backdoor('nextpage', 'nextpage,i')
>>> t.to_html(ctx)
>>> ctx.locals.i.set_backdoor('nextpage', 'nextpage,i')
>>> t.to_html(ctx)
>>> ctx.flush_content()
pagesize has_prevpage has_nextpage index start count value
----------------------------------------------------------
       3        False         True     0     0     0     0
       3        False         True     1     0     1     1
       3        False         True     2     0     2     2
pagesize has_prevpage has_nextpage index start count value
----------------------------------------------------------
       3         True         True     3     3     0     3
       3         True         True     4     3     1     4
       3         True         True     5     3     2     5
pagesize has_prevpage has_nextpage index start count value
----------------------------------------------------------
       3         True        False     6     6     0     6
       3         True        False     7     6     1     7
       3         True        False     8     6     2     8
```

`cols="..."` attribute

This attribute is used to format a sequence as multiple columns. The attribute value is an integer that specifies the number of columns.

Rather than evaluate the enclosed content once for each item in the sequence, the tag evaluates the content for each *row of items* in the sequence. The items in each row can be formatted by using an inner `<al-for>` tag.

By default the items flow down columns. To flow across columns you must use the `flow` (*flow="..." attribute*)

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-for iter="i" expr="range(15)" cols="4">
...  <al-for iter="c" expr="i.value()">
...   <al-value expr="' %2d' % c.value()">
...  </al-for whitespace>
... </al-for>
... ''').to_html(ctx)
>>> ctx.flush_content()
  0  4  8 12
  1  5  9 13
  2  6 10 14
  3  7 11
```

Multi-column formatting does not support pagination.

`continue` attribute

When paginating items via the `pagesize` (*pagesize="..." attribute*) attribute, the iterator index will reset to the first index displayed on the page if you use an iterator more than once on the page. The `continue` attribute suppresses the sequence index reset causing the elements to flow on from the previous page.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext, albatross.HiddenFieldSessionMixin):
...     def __init__(self):
...         albatross.SimpleContext.__init__(self, '.')
...         albatross.HiddenFieldSessionMixin.__init__(self)
...
```

```
>>> ctx = Ctx()
>>> albatross.Template(ctx, '<magic>', '''
... A:<al-for iter="i" expr="range(500)" pagesize="20" whitespace="indent">
...  <al-value expr="i.value()">
... </al-for whitespace>
... B:<al-for iter="i" pagesize="10" whitespace="indent">
...  <al-value expr="i.value()">
... </al-for whitespace>
... C:<al-for iter="i" pagesize="15" continue whitespace="indent">
...  <al-value expr="i.value()">
... </al-for whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
A: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
B: 0 1 2 3 4 5 6 7 8 9
C: 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

`expr="..."` attribute

The `expr` attribute specifies an expression that yields a sequence that the iterator specified in the `iter` (*iter="..." attribute*) attribute will iterate over. All of the enclosed content is then evaluated for each element in the sequence.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-for iter="i" expr="range(15)" whitespace="indent">
...  <al-value expr="i.value()">
... </al-for whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

`flow="..."` attribute

This attribute is used with the `cols` (*cols="..." attribute*) attribute to control the flow of values across columns. The default value is `"down"`. Use the value `"across"` to flow items across columns.

Rather than evaluate the enclosed content once for each item in the sequence, the tag evaluates the content for each *row of items* in the sequence. The items in each row can be formatted by using an inner `<al-for>` tag.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-for iter="i" expr="range(15)" cols="4" flow="across">
...  <al-for iter="c" expr="i.value()">
...   <al-value expr="' %2d' % c.value()">
...  </al-for whitespace>
... </al-for>
... ''').to_html(ctx)
>>> ctx.flush_content()
  0  1  2  3
  4  5  6  7
  8  9 10 11
 12 13 14
```

Multi-column formatting does not support pagination.

`iter="..."` attribute

This attribute specifies the name of the `ListIterator` (*ListIterator Objects*) that will be used to iterate over the items in the sequence defined by the expression in the `expr` (*expr="..." attribute*) attribute.

`pagesize="..."` attribute

This attribute is used to present a sequence of data one page at a time. The attribute value must be an integer that specifies the number of items to display in each page.

Use of the `pagesize` attribute places the sequence iterator into page mode and limits the number of elements that will be displayed.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext, albatross.HiddenFieldSessionMixin):
...     def __init__(self):
...         albatross.SimpleContext.__init__(self, '.')
...
...
>>> ctx = Ctx()
>>> ctx.locals.__dict__.keys()
[]
>>> albatross.Template(ctx, '<magic>', '''
... <al-for iter="i" expr="range(500)" pagesize="20" whitespace="indent">
...  <al-value expr="i.value()">
... </al-for whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
>>> ctx.locals.__dict__.keys()
['i']
```

Pagination support requires that session support be present in the execution context. All of the Albatross application objects provide session capable execution contexts by default. The `SimpleContext` class does not support sessions so it is necessary to augment the class for the above example. Note also that when the `<al-for>` tag processes the `pagesize` attribute it places the sequence iterator into the session.

`prepare` attribute

This attribute allows you to place pagination controls before the formatted sequence content.

When the `prepare` attribute is present the `<al-for>` tag will perform all processing but will not write any output. This allows you to test pagination results before presenting output.

For example:

```
>>> import albatross
>>> class Ctx(albatross.SimpleContext, albatross.HiddenFieldSessionMixin):
...     def __init__(self):
...         albatross.SimpleContext.__init__(self, '.')
...         albatross.HiddenFieldSessionMixin.__init__(self)
...
>>> ctx = Ctx()
>>> albatross.Template(ctx, '<magic>', '''
... <al-for iter="i" expr="range(500)" pagesize="20" prepare/>
... <al-if expr="i.has_prevpage()"> prev</al-if>
... <al-if expr="i.has_nextpage()"> next</al-if>
... <al-for iter="i" pagesize="20" whitespace="indent">
...  <al-value expr="i.value()">
... </al-for whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
 next 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Note the XML empty tag syntax on the `<al-for prepare>` tag.

`vars="..."` attribute

If this attribute is set, the current value of the iterator (as returned by `value()` will be saved to a variable of this name in the local namespace.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-for vars="v" expr="range(15)" whitespace="indent">
...  <al-value expr="v">
... </al-for whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

If the attribute is set to a comma separated list of variables, the iterator value will be unpacked into these variables. The iterator values must iterable in this case (typically a tuple or list). If there are more variables listed than there are values to be unpacked, then the unused variables are left unchanged. Conversely, if there are more values than variables, only the values with corresponding names will be unpacked.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.items = [(1.23, 'Red'), (4.71, 'Green'), (0.33, 'Blue')]
>>> albatross.Template(ctx, '<magic>', '''
... <al-for vars="price, label" expr="items">
...  <al-value expr="'$%0.2f' % price"> <al-value expr="label" whitespace>
... </al-for>
... ''').to_html(ctx)
>>> ctx.flush_content()
$1.23 Red
$4.71 Green
$0.33 Blue
```

ListIterator Objects

The iterator named in the `iter` (*iter="..." attribute*) attribute of the `<al-for>` tag is an instance of this class. By using an object to iterate over the sequence the toolkit is able to provide additional data that is useful in formatting HTML.

The iterator will retrieve each value from the sequence exactly once. This allows you to use objects that act as sequences by implementing the Python sequence protocol. Only `__getitem__()` is required unless you use pagination, then `__len__()` is also required.

`pagesize``()`
    Returns the pagesize that was set in the `pagesize` attribute.

`has_prevpage``()`
    Returns `TRUE` if the page `start` index is greater than zero indicating that there is a previous page.

has_nextpage()
: Returns TRUE if the sequence length is greater than the page start index plus the _pagesize member indicating that there is a next page.

  If the iterator has not been placed into "page mode" by the presence of a pagesize attribute a ListIteratorError exception will be raised.

index()
: Returns the index of the current sequence element.

start()
: Returns the index of the first sequence element on the page.

count()
: Returns the index of the current sequence element within the current page. This is equivalent to index() - start().

value()
: Returns the current sequence element.

Most of the methods and all of the members are not meant to be accessed from your code but are documented below to help clarify how the iterator behaves.

_index
: Current sequence index — returned by index().

_start
: Sequence index of first element on page — returned by start().

_count
: Sequence index of current element on page — returned by count().

_seq
: Sequence being iterated over — initialised to None and set by set_sequence().

_have_value
: Indicates the state of the current element. There are three possible values: None indicates that the state is unknown and will be established when the sequence is next accessed, zero indicates that the end of sequence has been reached and there is no valid element, and one indicates the current element is valid.

_pagesize
: Current page size — initialised to 0 and set by the presence of a pagesize attribute in the <al-for> tag.

__getstate__()
: When "page mode" is enabled the iterator is saved into the session (via the execution context add_session_vars() method). This restricts the Python pickler to saving only the _start and _pagesize members.

__setstate__(*tup*)
: Restores an iterator from the Python pickler.

__len__()
: When in "page mode" it returns the _pagesize member else it returns the length of the sequence.

set_backdoor(*op, value*)
: The <al-input> and <al-a> tags provide nextpage and prevpage attributes that generate names using a special backdoor format. When the browser request is merged the set_value() method of the NamespaceMixin directs list backdoor input fields to this method. Refer to the documentation in section *NamespaceMixin Class*.

  The *value* argument is the browser submitted value for the backdoor field. If a value was submitted for the backdoor field then the *op* argument is processed. If *op* equals "prevpage" or "nextpage" then the iterator selects the previous or next page respectively.

get_backdoor(*op*)
When generating backdoor fields for the `<al-input>` and `<al-a>` tags the toolkit calls this method to determine the value that will assigned to that field. The method returns `1`.

set_pagesize(*size*)
Sets the `_pagesize` member to *size*.

has_sequence()
Returns whether or not a sequence has been placed into the iterator (`_seq` is not `None`).

set_sequence(*seq*)
Sets the `_seq` to *seq*.

reset_index()
If the `<al-for>` tag does not contain a `continue` attribute then this is called just before executing the tag content for the first element in the sequence. It sets the `_index` member to `_start`.

reset_count()
This is called just before executing the tag content for the first element in the sequence. It sets the `_count` member to zero.

clear_value()
Sets the `_have_value` member to `None` which causes the next call of `has_value()` to retrieve the sequence element indexed by `_index`.

has_value()
When the `_have_value` member is `None` this method tries to retrieve the sequence element indexed by `_index`. If an element is returned by the sequence it is saved in the `_value` member and `_have_value` is set to one. If an `IndexError` exception is raised by the sequence then `_have_value` is set to zero.

The method returns TRUE if a sequence member is contained in `_value`.

By this mechanism the iterator retrieves each value from the sequence exactly once.

next()
Retrieves the next value (if available) from the sequence into the `_value` member.

set_value(*value*)
A back door hack for multi-column output that sets respective values of the iterator to sequences created by slicing the sequence in the `expr` attribute of the `<al-for>` tag.

8.4.9 `<al-lookup>`

The `<al-lookup>` tag uses a dictionary-style lookup to choose one of the contained `<al-item>` HTML fragments. If no `<al-item>` tag matches, then the tag returns any content that was not enclosed by an `<al-item>` tag. The `<al-item>` key values are derived by evaluating their `expr` attribute.

The `<al-lookup>` element will either be expanded in place if an `expr` attribute is given or, if named with an `name="..."` attribute, expanded later via an `<al-value> lookup="..."` tag.

`expr="..."` attribute

If the `expr` attribute is used, this is evaluated and the content of the matching `<al-item>` element is returned. If no match occurs, the unenclosed content is returned.

This form of the tag is akin to the switch or case statements that appear in some languages.

`name="..."` attribute

If the `name="..."` is used, the tag becomes a named lookup and expansion is deferred until the lookup is referenced via the `<al-value>` element. In this case, the lookup is performed on the evaluated value of the `<al-value>` `expr` attribute.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> MILD, MEDIUM, HOT = 0, 1, 2
>>> albatross.Template(ctx, '<magic>',
... '''<al-lookup name="curry">
... <al-item expr="MILD">Mild <al-value expr="curry"></al-item>
... <al-item expr="MEDIUM">Medium <al-value expr="curry"></al-item>
... <al-item expr="HOT">Hot <al-value expr="curry"></al-item>
... </al-lookup>''').to_html(ctx)
>>> ctx.locals.spicy = 2
>>> ctx.locals.curry = 'Vindaloo'
>>> albatross.Template(ctx, '<magic>', '''
... <al-value expr="spicy" lookup="curry" whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
Hot Vindaloo
```

By placing lookup tables in separate template files you can eliminate redundant processing via the `run_template_once()` execution context method. This method is defined in the `AppContext` class that is used as a base for all application execution contexts.

As the above example demonstrates, you are able to place arbitrary template HTML inside the lookup items. As the content of the item is only executed when referenced, all expressions are evaluated in the context of the template HTML that references the item.

8.4.10 `<al-item>`

The `<al-item>` tag must only be used as a child tag of an `<al-lookup>` (*<al-lookup>*) tag. to allow internal application values to be converted to display form.

`expr="..."` attribute

The `expr` attribute defines an expression that is evaluated to generate a lookup table key for the parent `<al-lookup>` (*<al-lookup>*) tag. When the parent `<al-lookup>` is executed all of the `expr` expressions are evaluated to build a dictionary of items.

For example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.key = 2
>>> albatross.Template(ctx, '<magic>',
... '''<al-lookup name="look">
... <al-item expr="1">item expr="1" key is <al-value expr="key"></al-item>
... <al-item expr="key">item expr="key" key is <al-value expr="key"></al-item>
... </al-lookup>''').to_html(ctx)
>>> ctx.locals.key = 1
>>> t = albatross.Template(ctx, '<magic>', '''
... <al-value expr="key" lookup="look" whitespace>
... ''')
>>> t.to_html(ctx)
>>> ctx.flush_content()
item expr="1" key is 1
>>> ctx.locals.key = 2
>>> t.to_html(ctx)
>>> ctx.flush_content()
item expr="key" key is 2
```

8.4.11 `<al-tree>`

This tag is used to display tree structured data. A pre-order traversal (node visited before each child) of the tree is performed and the enclosed content is evaluated for each node in the tree.

The `treefold`, `treeselect`, and `treeellipsis` attributes on the `<al-input>` (*<al-input>*) tag allow the user to open, close, and select lazy tree nodes.

Section *Displaying Tree Structured Data* contains an example of a simple usage of the `<al-tree>` tag.

The `samples/tree/tree1` sample program takes the simple example further and implements an application that places a checkbox next to the name of each node name. A unique input field name is generated for each checkbox by using the `alias` attribute described in section *<al-input>*.

The `samples/tree/tree2` sample program demonstrates the use of the `lazy` `<al-tree>` attribute that enables lazy child loading mode.

The `samples/tree/tree3` sample program demonstrates the use of the `ellipsis` `<al-tree>` attribute.

`ellipsis` attribute

The `ellipsis` attribute extends the `lazy` (*lazy attribute*) traversal. It collapses nodes close to the root of tree as deeper nodes are opened.

`expr="..."` attribute

The `expr` attribute defines the root of the tree traversal.

`iter="..."` attribute

This attribute specifies the name of the `TreeIterator` (*TreeIterator Objects*) that will be used to iterate over the nodes in the tree defined by the expression in the `expr` (*expr="..." attribute*) attribute.

`lazy` attribute

The `lazy` attribute allows lazy traversal of the tree, with child nodes only being loaded when their parent is open.

`single` attribute

The `single` attribute places the tree in single select mode. Whenever a new node is selected by browser input the previous selected node(s) will be deselected.

TreeNode Objects

There is no actual `TreeNode` class. Trees can be constructed from objects of any class as long as they implement the interface described here.

`children`

This member must only be present in non leaf nodes. It contains a list of immediate child nodes of this node.

When using lazy loaded trees this member should be initialised to an empty list in the constructor. The presence of the `children` member makes the toolkit treat the node as a non-leaf node.

`children_loaded`
This member needs only be present in non leaf nodes that are referenced by an `<al-tree>` tag in lazy mode. It should be initialised to `0` in the node constructor. The toolkit will set the member to `1` once it has called the `load_children()` method of the node.

`load_children``(`*ctx*`)`
This method must be defined for nodes that are referenced by an `<al-tree>` tag in lazy mode. It should must populate the `children` member with the immediate child nodes.

The toolkit will call this method when it needs to display the child nodes of a node and they have not yet been loaded (`children_loaded== 0`).

The toolkit "knows" when it needs to see the child nodes of a particular node so it asks that node to load the children. This allows potentially huge trees to be browsed by having the toolkit only load those nodes that are visible.

`albatross_alias``()`
This method must be defined for nodes that are referenced by an `<al-tree>` tag in lazy mode. It must return a unique string identifier for this node that is suitable for use as part of an HTML input field name or URL component via the special `treeselect`, `treefold` and `treeellipsis` attributes. The identifier must be the same each time the program is run (so `str(id(self))` will not work).

The `TreeIterator` uses the node identifier to record which nodes are open and which are selected. The same identifier is also used when the node is referenced via an `alias` (*alias="..." attribute*) attribute of an `<al-input>` tag.

TreeIterator Objects

An instance of `TreeIterator` class (or the sub-classes `LazyTreeIterator` (*LazyTreeIterator Objects*) or `EllipsisTreeIterator` (*EllipsisTreeIterator Objects*)) will be placed into the execution context using the name specified in the `iter` (*iter="..." attribute*) attribute. This iterator will contain traversal data for the current node each time the tag content is executed.

Note that it is also acceptable to create an instance of one of the TreeIterator classes prior to rendering the template. The `set_selected_aliases()` or `set_open_aliases()` methods can then be used to render the tree with nodes already selected or open.

By using an object to iterate over the tree the toolkit is able to provide additional data that is useful in formatting HTML. The toolkit also places the iterator into the session (you must be using an application class that supports sessions).

`value``()`
Returns the current node.

`tree_depth``()`
Returns the depth of the visible tree, from the root to deepest node. A single node tree has a depth of one.

`depth``()`
Returns the depth of the current node.

`span``()`
Is shorthand for `n.tree_depth() - n.depth()`. It is intended to be used for the `colspan` of the table cell containing the node name when laying the tree out in a table. See the `samples/tree/tree2.html` template for just such an example.

`line``(`*depth*`)`
Only useful when displaying a tree in tabular form where the root is in the first column of the first row. Returns the type of line that should be displayed in each column up to the depth of the current node.

A return value of `0` indicates no line, `1` indicates a line that joins a node later than this node, and `2` indicates a line that terminates at this node.

The example in section *Displaying Tree Structured Data* uses this method.

is_open()
: Returns TRUE if the current node is open. For non-lazy iterators, this is always TRUE except on leaf nodes.

is_selected()
: For non-lazy iterators, this always returns FALSE.

has_children()
: Returns TRUE if the current node has children (ie. it defines a `children` member).

Most of the methods and all of the members are not meant to be accessed from your code but are documented below to help clarify how the iterator behaves.

_value
: Stores a reference to the current tree node — returned by `value()`.

_stack
: As the tree is being traversed this list attribute records all parent nodes between the current node and the root. This is used to determine which branch lines should be drawn for the current node.

_line
: Stores the branch line drawing information for the current node. Elements of this list are returned by `line(depth)()`.

_tree_depth
: Stores the depth of the tree from the root to the deepest visible node. A single node tree has a depth of 1. This is calculated immediately before the tree is displayed. This is returned by `tree_depth()`.

set_line(*line*)
: Saves the *line* argument in `_line`.

set_value(*node*)
: Sets the `_value` to the *node* argument.

node_is_open(*ctx, node*)
: Called internally whenever the toolkit needs to determine the open state of a tree node. For non-lazy iterators, it returns whether or not the node in the *node* argument has children (because non-lazy iterators are always open).

LazyTreeIterator Objects

`<al-tree>` tags that include the `lazy` attribute use an instance of the `LazyTreeIterator` class. This class supports all the methods of the `TreeIterator` class, as well as the following:

is_open()
: Returns TRUE is the current node is open. Calls the `albatross_alias()` method of the current node and returns TRUE if the returned alias exists in the `_open_aliases` dictionary member.

is_selected()
: Returns TRUE if the current node is selected. Calls the `albatross_alias()` method of the current node and returns TRUE if the returned alias exists in the `_selected_aliases` dictionary member.

Some methods are designed to be called from application code, not from templates.

close_all()
: Closes all tree nodes by reinitialising the `_open_aliases` to the empty dictionary.

deselect_all()
: Deselects all tree nodes by reinitialising the `_selected_aliases` to the empty dictionary.

get_selected_aliases()
: Returns a sorted list of aliases for all nodes that are selected (ie. in the `_selected_aliases` member).

set_selected_aliases(*aliases*)
: Builds a new `_selected_aliases` member from the sequence of aliases passed in the *aliases* argument.

get_open_aliases()
Returns a sorted list of aliases for all nodes that are open (ie. in the `_open_aliases` member).

set_open_aliases(*aliases*)
Builds a new `_open_aliases` member from the sequence of aliases passed in the *aliases* argument.

`LazyTreeIterator` instances add the follow private methods and members:

_key
This member caches the value returned by the `albatross_alias()` method for the current node. This key is then used to look up the `_open_aliases` and `_selected_aliases` members.

_open_aliases
A dictionary that contains the aliases for all tree nodes that are currently open. The contents of this dictionary is maintained via the `set_backdoor()` method.

_selected_aliases
A dictionary that contains the aliases for all tree nodes that are currently selected. The contents of this dictionary is maintained via the `set_backdoor()` method.

__getstate__()
Used to save the iterator in the session. This restricts the Python pickler to saving only the `_lazy`, `_open_aliases` and `_selected_aliases` members.

__setstate__(*tup*)
Restores an iterator from the Python pickler.

set_value(*node*)
Sets the `_value` to the *node* argument. When operating in lazy mode the `albatross_alias()` method is called for *node* and the result is cached in `_key`.

node_is_open(*ctx, node*)
Called internally whenever the toolkit needs to determine the open state of a tree node. It returns whether or not the node in the *node* argument is open. This always returns `0` for leaf nodes as they do not have children.

When in lazy mode the open state of *node* is retrieved from `_open_aliases`. If the node state is open then the method checks the value of the node `children_loaded` member. If `children_loaded` is `FALSE` then the node `load_children()` is called to load the children of *node*.

set_backdoor(*op, key, value*)
The `<al-input>` and `<al-a>` tags provide `treefold` and `treeselect` attributes that generate names using a special backdoor format. When the browser request is processed, the `set_value()` method of the `NamespaceMixin` directs tree backdoor input fields to this method. Refer to the documentation in section *NamespaceMixin Class*.

When the *op* argument is `"treeselect"` the `_selected_aliases` is updated for the node identified by the *key* argument. If *value* is `FALSE` the key is removed else it is added.

When the *op* argument is `"treefold"` and *value* argument is `TRUE` then the open state of the node identified by the *key* argument is toggled.

get_backdoor(*op, key*)
When generating backdoor fields for the `<al-input>` and `<al-a>` tags the toolkit calls this method to determine the value that will assigned to that field.

When *op* is `"treeselect"` the method returns the current selected state of the node identified by *key*.

When *op* is `"treefold"` the method returns `1`.

EllipsisTreeIterator Objects

`EllipsisTreeIterator` objects are created by using the `ellipsis` attribute on an `<al-tree>` tag. Ellipsis trees are a variant of lazy trees where nodes at shallower levels are progressively collapsed into ellipses as the user opens deeper nodes. The user can reopen the collapsed nodes by selecting an ellipsis.

They support all the methods of the `LazyTreeIterator` (*LazyTreeIterator Objects*), as well as the following methods:

`node_type``()`
Returns `0` for a regular node, or `1` for nodes that have been collapsed into an ellipsis. This is actually implemented on the `LazyTreeIterator`, but will always return `0` there.

`EllipsisTreeIterator` objects also have the following private methods and members:

`_noellipsis_alias`
Records the last ellipsis to be selected by the user, and is used to suppress the generate of an ellipsis at that location next time the tree is rendered.

`node_use_ellipsis``(`*ctx, node*`)`
Returns `TRUE` if it is acceptable to render the specified `node` as an ellipsis. If the node's alias matches `_noellipsis_alias`, `FALSE` is return, otherwise `TRUE` is returned if any of the node's children are open.

The behaviour of the `set_backdoor()` and `get_backdoor()` methods has been extended to recognise a `treeellipsis` op. This is used to process browser requests to open an ellipsis (it sets the `_noellipsis_alias` member.

8.5 Macro Processing

Tags in this section provide a simple macro processing environment for template files.

The main purpose of Albatross macros is to provide a mechanism to divide your HTML into presentation structure and presentation appearance. By defining appearance presentation tricks inside macros you can make global changes to your web application appearance by changing one macro.

The `<al-macro>` (*<al-macro>*) and `<al-usearg>` (*<al-usearg>*) tags are used to define macros, while `<al-expand>` (*<al-expand>*) and `<al-setarg>` (*<al-setarg>*) are used to invoke and expand previously defined macros.

The `ResourceMixin` (*ResourceMixin Class*) and `ExecuteMixin` (*ExecuteMixin Class*) classes provide the Albatross macro definition and execution facilities respectively.

8.5.1 `<al-macro>`

The `<al-macro>` tag is used to define a macro. All enclosed content becomes part of the macro definition.

Executing the macro registers the macro with the execution context via the `register_macro()` method using the name in the `name` (*name="..." attribute*) attribute.

Note that the execution of the macro content is deferred until later when the macro is expanded via the `<al-expand>` (*<al-expand>*) tag. This means that executing a macro definition produces no output. Output is produced only when the macro is expanded.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="noargs">
...  Will be executed when macro is expanded.
... </al-macro>
... ''').to_html(ctx)
>>> ctx.flush_content()
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="noargs"/>
... ''').to_html(ctx)
>>> ctx.flush_content()
Will be executed when macro is expanded.
```

The deferred execution also means that you can include content that only works within the context of the `<al-expand>` tag.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="oops">
...  <al-value expr="oops">
... </al-macro>
... ''').to_html(ctx)
>>> ctx.flush_content()
>>> templ = albatross.Template(ctx, '<magic>', '''
... <al-expand name="oops"/>
... ''')
>>> try:
...     templ.to_html(ctx)
...     ctx.flush_content()
... except NameError, e:
...     print e
...
name 'oops' is not defined
>>> ctx.locals.oops = 'there is now'
>>> templ.to_html(ctx)
>>> ctx.flush_content()
there is now
```

In the above example the content of the macro makes reference to a *oops* that is not defined in the execution context when the macro was defined.

Inside a macro definition you can use as yet undefined macros.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="bold-red">
... <font color="red">more <al-expand name="bold"><al-usearg></al-expand> please</font>
... <font color="red"><al-expand name="bold">more <al-usearg> please</al-expand></font>
... </al-macro>
...
... <al-macro name="bold">
...  <b><al-usearg></b></al-macro>
... ''').to_html(ctx)
>>> ctx.flush_content()
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="bold-red">spam</al-expand>
... ''').to_html(ctx)
>>> ctx.flush_content()
<font color="red">more <b>spam</b> please</font>
<font color="red"><b>more spam please</b></font>
```

Care must by taken to ensure that you do not make circular macro references else you will cause a stack overflow.

`name="..."` attribute

The `name` attribute is used to uniquely identify the macro in the application.

8.5.2 `<al-usearg>`

The `<al-usearg>` tag is used inside a macro definition to define the location where content enclosed by the `<al-expand>` (*<al-expand>*) tag should be placed when the macro is expanded.

All content enclosed by the `<al-expand>` tag is passed to the macro as the unnamed argument. The unnamed argument is retrieved in the macro definition by using an `<al-usearg>` tag without specifying a `name` (*name="..." attribute*) attribute. When a macro expects only one argument it is best to use this mechanism.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="double">
...  1. <al-usearg>
...  2. <al-usearg>
... </al-macro>
... ''').to_html(ctx)
>>> ctx.flush_content()
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="double">
...  spam
... </al-expand>''').to_html(ctx)
>>> ctx.flush_content()
1. spam
2. spam
```

`name="..."` attribute

Macros can be defined to accept multiple arguments. The `name` attribute is used to retrieve named arguments. When invoking a macro that accepts named arguments the `<al-setarg>` (*<al-setarg>*) tag and `name` attribute are used to define the content for each named argument.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="namedargs" whitespace="indent">
...  The unnamed arg (<al-usearg>) still works,
...  but we can also include named arguments (<al-usearg name="arg1">)
... </al-macro>
... ''').to_html(ctx)
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="namedargs">
...  unnamed arg<al-setarg name="arg1">
...  named arg: arg1</al-setarg>
... </al-expand>
... ''').to_html(ctx)
>>> ctx.flush_content()
 The unnamed arg (unnamed arg) still works,
 but we can also include named arguments (named arg: arg1)
```

8.5.3 `<al-setdefault>`

The `<al-setdefault>` tag is used inside a macro definition to specify default content for a named macro argument. The content enclosed by this tag will be used if the caller does not override it with a `<al-setarg>` tag.

Note that only named arguments can have a default as the unnamed argument is always set, implicitly or explicitly, by the calling `<al-expand>` tag.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="pagelayout">
...  <al-setdefault name="title">Parrot</al-setdefault>
...  <title><al-usearg name="title"></title>
... </al-macro>
... ''').to_html(ctx)
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="pagelayout">
...  <al-setarg name="title">Lumberjack</al-setarg>
... </al-expand>''').to_html(ctx)
>>> ctx.flush_content()
<title>Lumberjack</title>
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="pagelayout">
... </al-expand>''').to_html(ctx)
>>> ctx.flush_content()
<title>Parrot</title>
```

name="..." attribute

The name attribute is used to identify the named macro argument that will receive the enclosed content.

8.5.4 <al-expand>

The <al-expand> tag is used to expand a previously defined macro.

You can pass macro expansions as arguments to other macros.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="red">
... <font color="red"><al-usearg></font>
... </al-macro>
...
... <al-macro name="bold"><b><al-usearg></b></al-macro>
... ''').to_html(ctx)
>>> ctx.flush_content()
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="red">more <al-expand name="bold">spam</al-expand> please</al-expand>
... <al-expand name="red"><al-expand name="bold">more spam please</al-expand></al-expand>
... ''').to_html(ctx)
>>> ctx.flush_content()
<font color="red">more <b>spam</b> please</font>
<font color="red"><b>more spam please</b></font>
```

All arguments to macros are executed each time they are used in the macro definition. This means that you need to be aware of side effects when using arguments more than once inside a macro.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="yummy">
...  <al-for iter="i" expr="range(3)">
...   <al-usearg whitespace="indent">
...  </al-for>
... </al-macro>
... ''').to_html(ctx)
>>> ctx.locals.food = 'spam'
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="yummy">
...  <al-exec expr="food = food + '!'">
...  <al-value expr="food">
... </al-expand whitespace>
... ''').to_html(ctx)
>>> ctx.flush_content()
spam! spam!! spam!!!
```

`name="..."` attribute

The `name` attribute contains the name of the macro that will be expanded. Macros are defined and names using the `<al-macro>` (*<al-macro>*) tag.

`...arg="..."` attributes

When macro arguments are simple strings, they can be specified as `<al-expand>` attributes by appending `arg` to the argument name. So, to set an argument called `title`, you could add an `titlearg` attribute to the `<al-expand>` tag.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="pagelayout">
...  <title><al-usearg name="title"></title>
... </al-macro>
... ''').to_html(ctx)
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="pagelayout" titlearg="Lumberjack" />''').to_html(ctx)
>>> ctx.flush_content()
<title>Lumberjack</title>
```

If the macro argument is longer or needs to contain markup, the `<al-setarg>` (*<al-setarg>*) tag should be used instead.

`...argexpr="..."` attributes

Macro arguments can also be derived by evaluating a python expression. Attributes of the `<al-expand>` tag that end in `argexpr` are evaluated, and the base name becomes the macro argument of that name.

For example:

```
<al-expand name="pagelayout" titleargexpr="foo" />
```

is functionally equivilent to:

```
<al-expand name="pagelayout">
    <al-setarg name="title"><al-value expr="foo"></al-setarg>
</al-expand>
```

For a more complete example:

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> ctx.locals.title = 'Lumberjack'
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="pagelayout">
...  <title><al-usearg name="title"></title>
... </al-macro>
... ''').to_html(ctx)
```

```
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="pagelayout" titleargexpr="title" />''').to_html(ctx)
>>> ctx.flush_content()
<title>Lumberjack</title>
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="pagelayout">
...  <al-setarg name="title"><al-value expr="title"></al-setarg>
... </al-expand>''').to_html(ctx)
>>> ctx.flush_content()
<title>Lumberjack</title>
```

8.5.5 `<al-setarg>`

The `<al-setarg>` tag is used pass content to a macro. All content enclosed by the tag will be passed as an argument to the macro named by the parent `<al-expand>` (*<al-expand>*) tag.

The `<al-setarg>` tag is normally used to pass content to macros that define named arguments, but can also be used to enclose the unnamed argument.

```
>>> import albatross
>>> ctx = albatross.SimpleContext('.')
>>> albatross.Template(ctx, '<magic>', '''
... <al-macro name="title">
...  <title><al-usearg></title>
... </al-macro>
... ''').to_html(ctx)
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="title">
...  <al-setarg>Lumberjack</al-setarg>
... </al-expand>''').to_html(ctx)
>>> ctx.flush_content()
<title>Lumberjack</title>
>>> albatross.Template(ctx, '<magic>', '''
... <al-expand name="title">
...  Lumberjack
... </al-expand>''').to_html(ctx)
>>> ctx.flush_content()
<title>Lumberjack
</title>
```

`name="..."` attribute

The `name` attribute is used to identify the named macro argument that will receive the enclosed content.

CHAPTER

NINE

DEVELOPING CUSTOM TAGS

In complex applications you may encounter presentation problems which the standard collection of tags cannot easily solve. Albatross allows you to register additional tags, which are then available for use in your templates. Custom tags are named with an `alx-` prefix to distinguish them from standard `al-` tags.

Custom tags should subclass either `EmptyTag` or `EnclosingTag`, and have a `name` class attribute. The name should start with `alx-` and contain only letters, numbers and the underscore character.

Custom tags that produce form inputs need to register the names of those inputs with the `NameRecorderMixin` via the `input_add()` method. For more information, see section *NameRecorderMixin*, NameRecorderMixin in the Mixin Class Reference.

The following is a simple calendar tag which formats a single month like the unix **cal(1)** program.

```
import time
import calendar
import albatross


class Calendar(albatross.EmptyTag):

    name = 'alx-calendar'

    def to_html(self, ctx):
        year = self.get_attrib('year')
        if year is not None:
            year = ctx.eval_expr(year)
        month = self.get_attrib('month')
        if month is not None:
            month = ctx.eval_expr(month)
        if month is None or year is None:
            now = time.localtime(time.time())
            if year is None:
                year = now[0]
            if month is None:
                month = now[1]
        ctx.write_content('<table>\n')
        ctx.write_content('<tr align="center"><td colspan="7">%s %s</td></tr>\n' \
                          % (calendar.month_name[month], year))
        ctx.write_content('<tr>')
        for i in range(7):
            ctx.write_content('<td>%s</td>' \
                              % calendar.day_abbr[(i + 6) % 7][:2])
        ctx.write_content('</tr>\n')
        calendar.setfirstweekday(6)
        for r in calendar.monthcalendar(year, month):
            ctx.write_content('<tr align="right">')
            for i in range(7):
                if r[i]:
```

```
                ctx.write_content('<td>%s</td>' % r[i])
            else:
                ctx.write_content('<td></td>')
        ctx.write_content('</tr>\n')
    ctx.write_content('</table>\n')
```

To use the tag in your application you must make the class available to the execution context. If you are using an Albatross application object you can do this by passing the class to the `register_tagclasses()` method of the application object.

```
from albatross import SimpleApp

app = SimpleApp('ext.py', '.', 'start')
app.register_tagclasses(Calendar)
```

All Albatross application classes inherit from the `ResourceMixin` in the `albatross.context` module. Execution contexts which are used with application objects inherit from the `AppContext` class from the `albatross.app` module which automatically retrieves all resources from the parent application object.

If you are using the `SimpleContext` class for your execution context then you will need to call the `register_tagclasses()` method of the execution context immediately after construction.

The following is an example template file which uses the `<alx-calendar>` tag.

```
<html>
<head><title>Calendar for <al-value expr="year"></title></head>
<body>
<h1>Calendar for <al-value expr="year"></h1>
<table cellpadding="10">
<al-for iter="r" expr="range(1,13)" cols="3" flow="across">
 <tr valign="top">
 <al-for iter="m" expr="r.value()">
  <td><alx-calendar month="m.value()" year="year"></td>
 </al-for>
 </tr>
</al-for>
</table>
</body>
</html>
```

A complete program which uses this extension tag and template file can be found in the `samples/extension` directory. Use the `install.py` script to install the sample.

```
cd samples/extension
python install.py
```

The implementation of the standard tags also makes a good reference when writing custom tags. All standard tags are defined in `albatross.tags`.

9.1 `albatross.template` — Base classes for implementing tags

The module contains the following classes which are intended to be used in implementing custom tags.

class `Tag` (*ctx, filename, line_num, attribs*)
    This is the base class upon which all tags are implemented. You are unlikely to ever subclass this directly. The `EmptyTag` and `EnclosingTag` classes inherit from this class.

class `EmptyTag` (*ctx, filename, line_num, attribs*)
    Use this class as a subclass for all tags which do not require a closing tag and therefore do not enclose content. Examples of standard HTML tags which do not enclose content are `<BR>` and `<HR>`.

class EnclosingTag (*ctx, filename, line_num, attribs*)
: Use this class as a subclass for all tags which enclose content. Examples of standard HTML tags which enclose content are `<BODY>` and `<TABLE>`.

class Text (*text*)
: A simple wrapper around the string passed in the *text* constructor argument which passes that string to the `to_html()` method when the object is converted to HTML.

class Content ()
: A simple wrapper around a list which calls the `to_html()` method of all list elements when the object is converted to HTML.

9.1.1 Tag Objects

`raise_error` (*msg*)
: Raises a `TemplateError` exception using the string in the *msg* argument.

`has_attrib` (*name*)
: Returns `TRUE` if the attribute specified in the *name* argument was defined for the tag. All attribute names are converted to lower case by the template parser.

`assert_has_attrib` (*name*)
: If the attribute specified in the *name* argument is not defined for the tag a `TemplateError` exception will be raised.

`assert_any_attrib` (**names*)
: If none of the attributes specified by the arguments are defined for the tag a `TemplateError` exception will be raised.

`get_attrib` (*name, [default ‘‘= None‘‘]*)
: Retrieves the value of the attribute specified in the *name* argument.

`set_attrib` (*name, value*)
: Sets the value of the attribute named in the *name* argument to the value in the *value* argument.

`set_attrib_order` (*order*)
: Defines the order that the tag attributes will be written during conversion to HTML. The template parser captures the attribute sequence from the template file then calls this method.

`attrib_items` ()
: Returns a list of attribute name, value tuples which are defined for the tag.

`write_attribs_except` (*ctx, [...]*)
: Sends all tag attributes to the `write_content()` method of the execution context in the *ctx* argument. Any attributes named in additional arguments will not be written.

9.1.2 EmptyTag Objects

`has_content` ()
: Returns `0` to inform the template parser that the tag does not enclose content.

`to_html` (*ctx*)
: The template interpreter calls this method to convert the tag to HTML for the execution context in the *ctx* argument. The default implementation does nothing.

: You must override this method in your tag class to perform all actions which are necessary to “execute” the tag.

9.1.3 EnclosingTag Objects

`content`
: An instance of the `Content` class which is created during the constructor.

has_content()
Returns `1` to inform the template parser that the tag encloses content.

append(*item*)
Called by the template parser to append the content in the *item* argument to the tag. The method implementation simply passes *item* to the `append()` method of the `content` member.

You should override this method if you need to maintain multiple content lists within your tag.

to_html(*ctx*)
The template interpreter calls this method to convert the tag to HTML for the execution context in the *ctx* argument. The default implementation passes *ctx* to the the `to_html()` method of the `content` member.

You must override this method in your tag class to perform all actions which are necessary to “execute” the tag.

9.1.4 Text Objects

to_html(*ctx*)
Sends the wrapped text to the `write_content()` method of the execution context in the *ctx* argument. You should not ever need to subclass these objects.

9.1.5 Content Objects

append(*item*)
Appends the value in the *item* argument to the internal Python list.

to_html(*ctx*)
Sequentially invokes the `to_html()` method of every item in the internal Python list passing the *ctx* argument.

CHAPTER

TEN

MIXIN CLASS REFERENCE

Most of Albatross exists as a collection of plug compatible mixin classes which you select from to define the way your application should behave and how it will be deployed. Figure *Toolkit Components* shows the organisation of the component types in the toolkit.

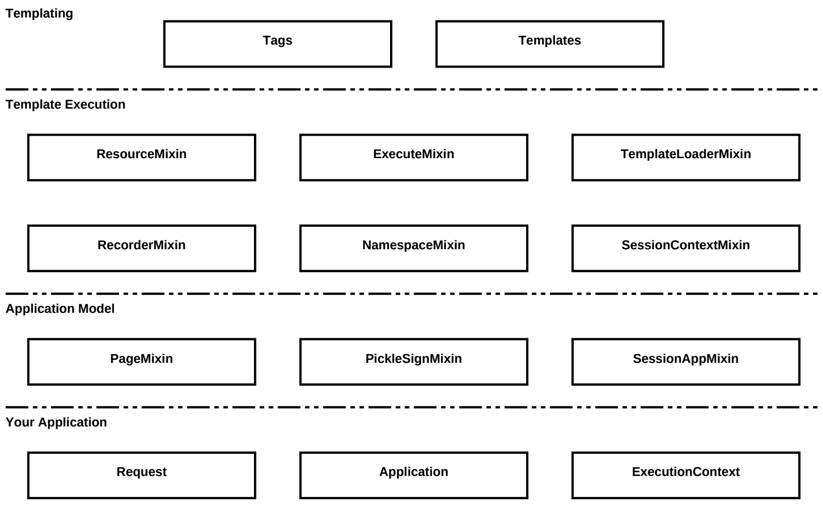


Figure 10.1: Toolkit Components

The divisions in the diagram represent conceptually different functional areas within the toolkit.

Templating This layer provides the Albatross templating functionality. Template classes make use of the methods defined in the next layer down to access application functionality and data.

Classes in this layer are defined in the `albatross.template` and `albatross.tags` modules.

Template Execution An execution context suitable for interpreting template files is constructed by combining one mixin of each type from this layer.

Application Model Execution contexts which subclass a `PageMixin` class in addition to the mixins from the above layer are suitable for use in Albatross applications. The `PageMixin` class controls how the application locates code and template files for each page served by the application. The `PickleSignMixin` class is responsible for modifying pickles which are sent to the browser to prevent or detect modification.

Your Application In this layer you will typically create your own application and execution context classes by subclassing a prepared application class.

For the most part Albatross applications are independent of the method by which they are deployed. Depending upon which `Request` class you choose from this layer you can either deploy your application via CGI, `mod_python` (http://www.modpython.org/), FastCGI (http://www.fastcgi.com/) or as a stand-alone python HTTP server.

Wherever possible the Albatross mixin classes use member names which begin with double underscore to trigger Python name mangling. This protects your classes from having member name clashes with private members of the mixin classes. Any member names which are not mangled are intended to be accessed in your application code.

10.1 ResourceMixin Class

Albatross only supplies one class for this function; the `ResourceMixin` class. This mixin manages application resources which which do not change regardless of context. The resources managed are tag classes, HTML macros, and HTML lookup tables.

The `SimpleContext` execution context class subclasses the `ResourceMixin` class. During the constructor it registers all of the standard Albatross tags. As HTML templates are executed the macros and lookup tables in those templates are registered.

All standard Albatross application classes inherit from the `Application` class which in turn subclasses the `ResourceMixin` class. During the application class constructor all of the standard Albatross tags are registered. The `AppContext` class which is subclassed by all Albatross application execution context classes proxies all HTML macro and lookup table methods and directs them to the application object.

`__init__`()
: When you inherit from the `ResourceMixin` class you must call this constructor to initialise the internal variables.

`get_macro`(*name*)
: Returns the macro previously registered by the name in the *name* argument. If no such macro exists `None` is returned.

`register_macro`(*name, macro*)
: Registers the HTML macro in the *macro* argument under the name in the *name* argument.

`get_lookup`(*name*)
: Returns the lookup table previously registered by the name in the *name* argument. If no such lookup exists `None` is returned.

`register_lookup`(*name, lookup*)
: Registers the HTML lookup table in the *lookup* argument under the name in the *name* argument.

`discard_file_resources`(*filename*)
: Discards macros and lookups loaded from *filename*. This is called prior to reloading a template.

`get_tagclass`(*name*)
: Returns the tag class in which the `name` member matches the name in the *name* argument. If no such tag class exists `None` is returned.

`register_tagclasses`(...)
: Registers one or more tag classes indexing them by the value in the `name` member of each class.

10.2 ExecuteMixin Class

Albatross only supplies one class for this function; the `ExecuteMixin` class. This mixin provides a “virtual machine” which is used to execute HTML template files.

All standard Albatross execution context classes inherit from this class.

`__init__`()
: When you inherit from the `ExecuteMixin` class you must call this constructor to initialise the internal variables.

`get_macro_arg`(*name*)
: Retrieves a macro argument from the macro execution stack. The stack is searched from the most recently pushed dictionary for an argument keyed by *name*. If no argument is found a `MacroError` exception is raised.

push_macro_args (*dict*)
Pushes a dictionary of macro arguments onto the macro execution stack.

pop_macro_args ()
Pops the most recently pushed dictionary of macro arguments from the macro execution stack.

push_content_trap ()
Saves accumulated content on the trap stack and then resets the content list.

pop_content_trap ()
Joins all parts on the content list and returns the result. Sets the content list to the value popped off the trap stack.

The content trap is useful for performing out-of-order execution of template text. The `<al-option>` tag makes use of the content trap.

write_content (*data*)
Appends the content in *data* to the content list.

flush_content ()
Does nothing if a content trap stack is in effect, otherwise it joins all parts in the content list and sends it to the browser via the `send_content()` method. The content list is then reset via the `reset_content()` method.

flush_html ()
This is an alias for `flush_content()`.

send_content (*data*)
Sends the content passed in the *data* argument to standard output. This is overridden in the `AppContext` class to redirect *data* to the `write_content()` method of the application referenced in the `app` member.

reset_content ()
Sets the content list to an empty list and clears the content trap stack.

10.3 ResponseMixin Class

The `ResponseMixin` class provides functionality to manage the delivery of the application response to the browser. The class maintains headers in a case insensitive ordered dictionary that ensures the headers are sent to the browser in the same sequence as they are set (via `set_header()` or `add_header()`).

The class automatically sends the headers to the browser when the first content is sent. Any attempt to modify or send headers after they have been sent will raise an `ApplicationError` exception.

All Albatross execution context classes except for `SimpleContext` inherit from this class.

__init__ ()
When you inherit from the `ResponseMixin` class you must call this constructor to initialise the internal variables.

get_header (*name*)
Returns a list of values of the *name* header from the internal store. If the header does not exist then `None` is returned.

set_header (*name, value*)
Sets the value of the *name* header to *value* in the internal store, replacing any existing headers of the same name.

If headers have already been sent to the browser then an `ApplicationError` exception will be raised.

add_header (*name, value*)
Sets the value of the *name* header to *value* in the internal store, appending the new header immediately after any existing headers of the same name.

If headers have already been sent to the browser then an `ApplicationError` exception will be raised.

del_header(*name*)
Removes the *name* header from the internal store.

If headers have already been sent to the browser then an `ApplicationError` exception will be raised.

write_headers()
Writes all headers in ascending sequence to the browser. Each header is sent via the `Request` object `write_header()` method. At the end of headers the `Request` object `end_headers()` method is called.

If headers have already been sent to the browser then an `ApplicationError` exception will be raised.

send_content(*data*)
Sends the content in *data* to the browser via the `Request` object `write_content()` method.

If headers have not already been delivered to the browser then the `write_headers()` method is called before the *data* is written.

send_redirect(*loc*)
If a cookie header has been set it is sent to the browser then the `redirect()` method of the `request` member is called and the result is returned.

10.4 TemplateLoaderMixin Classes

This mixin is responsible for loading template files. Albatross supplies two classes; `TemplateLoaderMixin` and `CachingTemplateLoaderMixin`.

10.4.1 TemplateLoaderMixin

The `TemplateLoaderMixin` class is a simplistic loader which performs no caching.

__init__(*base_dir*)
When you inherit from the `TemplateLoaderMixin` class you must call the constructor to define the root directory where template files will be loaded in the *base_dir* argument.

load_template(*name*)
Load and return the parsed template file specified in the *name* argument. The path to the template file is constructed by performing `os.path.join()` on the *base_dir* specified in the constructor and the *name* argument.

If there is an error reading the template a `TemplateLoadError` will be raised.

The class remembers the names of all loaded templates.

load_template_once(*name*)
Returns `None` if the template specified in the *name* argument has been previously loaded. If not previously loaded it is loaded via the `load_template()` method and returned.

10.4.2 CachingTemplateLoaderMixin

The `CachingTemplateLoaderMixin` class caches loaded templates to and only reloads them if they have been modified since they were last loaded.

__init__(*base_dir*)
When you inherit from the `CachingTemplateLoaderMixin` class you must call the constructor to define the root directory where template files will be loaded in the *base_dir* argument.

load_template(*name*)
Return the parsed template file specified in the *name* argument. The path to the template file is constructed by performing `os.path.join()` on the *base_dir* specified in the constructor and the *name* argument.

If there is an error reading the template a `TemplateLoadError` will be raised.

If the template has been previously loaded it will only be reloaded if it has been modified since last load.

load_template_once(*name*)
Call the `load_template()` method and return the template if it is either loaded for the first time or reloaded, else return `None`.

10.5 RecorderMixin Classes

This mixin is passed form and input field recording messages as `<al-form>`, `<al-input>`, `<al-select>`, and `<al-a>` tags are executed. Albatross supplies two classes; `StubRecorderMixin` and `NameRecorderMixin`.

10.5.1 StubRecorderMixin

The `StubRecorderMixin` class ignores all form events.

form_open()
Does nothing.

form_close()
Does nothing.

input_add(*itype, name, [value ``= None``]*)
Does nothing.

merge_request()
Merges request fields into `ctx.locals`.

10.5.2 NameRecorderMixin

The `NameRecorderMixin` class records details of all input fields used by a form. When the form element is closed, a hidden field named `__albform__` containing these details is added to the form.

When processing a request, the `merge_request()` method only merges fields with `ctx.locals` when they match the details found in the submitted `__albform__` field.

__init__()
When you inherit from the `NameRecorderMixin` class you must call the constructor.

form_open()
Called when the `<al-form>` tag is opened.

form_close()
Called just before the `<al-form>` tag is closed. A hidden field named `__albform__` is written to the output.

input_add(*itype, name, [value ``= None``], [return_list ``= 0``]*)
Called when an `<al-input>` tag is executed. The *itype* argument contains the `type` attribute from the input tag, *name* contains the `name` tag attribute, and *value* contains the value of the input field if it is known and relevant. The *return_list* argument indicates the presence of the `list` attribute on the input tag.

As fields are added to each form the value of the *return_list* argument is checked against any previous setting of the argument for the same field name. The argument value is also checked against whether or not there are multiple instances of the field name. An detected discrepancy between the argument value and actual fields will raise a `ApplicationError` exception.

merge_request()
Retrieves the `__albform__` value from the browser request decodes it and then merges the browser request into the local namespace accordingly.

If an input field has been flagged to return a list (via the `list` tag attribute) then the method will create a list in `ctx.locals` for the field regardless of the number of values sent by the browser. An empty list is created when the field is missing from the browser request.

Request fields not listed in `__albform__` are ignored.

10.6 NamespaceMixin Class

Albatross only supplies one class for this function; the `NamespaceMixin` class. This mixin provides a local and global namespace for evaluating expressions embedded in HTML template files.

When the browser request is merged into the execution context the input field values are written to the local namespace.

`__init__``()`
When you inherit from the `NamespaceMixin` class you must call the constructor.

The global namespace for evaluating Python expressions in HTML templates is initialised as an empty dictionary in the constructor.

`locals`
An empty object which is used for the local namespace for evaluating expressions in HTML templates. It is initialised as an instance of an empty class in the constructor to allow values to simply be assigned to attributes of this member.

Loading the session merges the session values into this member.

`clear_locals``()`
Resets the `locals` member to an empty object.

`set_globals``(`*dict*`)`
Sets the global namespace for evaluating expressions to the *dict* argument.

The `SimpleContext` class constructor automatically sets this to the globals of the function which invoked the `SimpleContext` constructor.

The `run_template()` and `run_template_once()` methods of the `AppContext` calls this method to set global namespace to the globals of the calling function.

`eval_expr``(`*expr*`)`
Called by the template file interpreter to evaluate the embedded Python expression in the *expr* argument.

`set_value``(`*name, value*`)`
Sets the local namespace attribute named in the *name* argument to the value in the *value* argument. If the *name* argument begins with an underscore the method will raise a `SecurityError` exception.

This is used by the application `merge_request()` method to merge individual browser request fields into the local namespace.

There is a special "backdoor" identifier format which which directs browser request fields to the `set_backdoor()` method of `ListIterator` and `TreeIterator` objects. The backdoor identifiers are generated by the `<al-input>` and `<al-a>` tags to implement sequence and tree browsing requests.

The method implements a parser which can handle names of the form:

```
name           ::=  identifier \| list-backdoor \| tree-backdoor
identifier     ::=  identifier ((“.” identifier) \| (“[” number “]”))\*
list-backdoor  ::=  operation “,” iter
tree-backdoor  ::=  operation “,” iter “,” alias
```

`merge_vars``(...)`
This method merges request fields matching a prefix given in the argument list to the local namespace (via the `set_value()` method described above).

Normally, merging of request fields is automatic: either all request fields are copied when `StubRecorderMixin` is used, or fields listed in `__albform__` are copied when `NameRecorderMixin` is used. However in cases where `NameRecorderMixin` is used and no `__albform__` field is present, request merging does not occur, and this method is needed to allow the application to explicitly request fields be merged.

`make_alias`(*name*)
Called to generate an alternate name for an object referenced in the `alias` attribute of an Albatross tag.

The method resolves the *name* argument up to the last "." and then calls the `albatross_alias()` method of the resolved object. The resolved object is then entered into the local namespace and the session using the name returned by `albatross_alias()`.

The return value is a new name by combining the name returned by `albatross_alias()` with the part of the original name following and including the last ".".

Refer to the `<al-input>` documentation in the *<al-input>* section for an explanation of why this method exists.

`get_value`(*name*)
Retrieves the value identified by the *name* argument from the local namespace. If the named value does not exist then `None` is returned.

`has_value`(*name*)
Returns whether or not the value named in the *name* attribute exists in the local namespace.

`has_values`(...)
Returns TRUE only if values named in the argument list exist in the local namespace.

10.7 SessionContextMixin Classes

This mixin is used to manage the encoding and decoding of session data. Albatross supplies a number of classes for use in the execution context; `StubSessionMixin`, `SessionBase`, `HiddenFieldSessionMixin`, `SessionServerContextMixin`, and `SessionFileContextMixin`. The `SessionBase` class provides base functionality for non-stub session classes.

Loading and storing of session data is usually performed by an application mixin.

The `SessionServerAppMixin` is designed to be used in the application object.

10.7.1 StubSessionMixin

The `StubSessionMixin` class ignores all session operations.

`add_session_vars`(...)
Does nothing.

`del_session_vars`(...)
Does nothing.

`encode_session`()
Does nothing.

`load_session`()
Does nothing.

`save_session`()
Does nothing.

`remove_session`()
Does nothing.

`set_save_session`(*flag*)
Does nothing.

should_save_session()
Returns `0`.

10.7.2 SessionBase

The `SessionBase` class provides base session handling functionality which is used by all standard Albatross execution context session mixin classes.

__init__()
When you inherit from the `SessionBase` class you must call this constructor.

The class maintains a dictionary of all names from the execution context local namespace which belong in the session. This dictionary is restored along with the session when the session is decoded.

add_session_vars(...)
Adds all listed names to the session dictionary. The named variables must exist in the `locals` member or a `SessionError` will be raised.

The names can optionally be supplied as a list or tuple of names.

default_session_var(*name, value*)
Adds a name to the session directory. Sets a value in the local namespace if the name is not already in the local namespace.

del_session_vars(...)
Deletes all listed names from the session dictionary.

The names can optionally be supplied as a list or tuple of names.

session_vars()
Returns a list of the names that are currently in the session.

remove_session()
Deletes all names from the session dictionary and clears all values from the local namespace via the `clear_locals()` method.

decode_session(*text*)
Performs `cPickle.loads()` to retrieve a dictionary of session values. The dictionary is merged into the session local namespace. Adds the keys of the dictionary to the session dictionary.

Note that an import hook is used around the `cPickle.loads()` to redirect requests to load page modules to the `app.load_page_module()` method. This allows the pickler to find classes which are defined in application page modules. Whether a module is a page module is determined by calling the `app.is_page_module()` method, passing the module name.

encode_session()
Builds a dictionary by extracting all local namespace values which are listed in the session dictionary. A test pickle is performed on each value and any unpickleable value is discarded and an error message is written to `sys.stderr`.

The dictionary is then passed to `cPickle.dumps()` and the result is returned.

set_save_session(*flag*)
Sets the flag which controls whether the session will be saved at the end of request processing. By default the internal flag is `TRUE` which means the session will be saved.

should_save_session()
Returns the flag which controls whether the session will be saved at the end of request processing.

10.7.3 HiddenFieldSessionMixin

Saves session state to a hidden field named `__albstate__` at the end of every form produced by `<al-form>` tags.

Inherits from the `SessionBase` class so you must call the constructor if you subclass this class.

`encode_session``()`
Extends the base class `encode_session()` method to `zlib.compress()` and `base64.encodestring()` the result. This makes the session data suitable for placing in a hidden field in the HTML.

`load_session``()`
This is called from the `Application` class `load_session()` method. The session state is retrieved from the browser request, decoded and decompressed then passed to the `decode_session()` method.

`save_session``()`
This is called from the `Application` class `save_session()` method at the end of the request processing sequence. The method does nothing because the session state is saved in hidden fields in the HTML.

`form_close``()`
Called just before the `<al-form>` tag is closed. If the session is flagged to be saved a hidden field named `__albstate__` is written to the output.

Note that this method is also present in the `RecorderMixin`, so if you inherit from the `HiddenFieldSessionMixin` class you must define a `form_close()` method in the derived class which calls this method in both of the super classes.

10.7.4 SessionServerContextMixin

This class works in concert with the `SessionServerAppMixin` application mixin class to store session data in the Albatross session server. All management of session data storage is performed by the application class.

Inherits from the `SessionBase` class so you must call the constructor if you subclass this class.

`__init__``()`
When you inherit from the `SessionServerContextMixin` class you must call this constructor.

`sesid``()`
Returns the session id.

`load_session``()`
This is usually called from the `Application` class `load_session()` method. Retrieves the session id and then either retrieves an existing session or creates a new session via the application object.

If an existing session is retrieved it is passed to `base64.decodestring()` and `zlib.decompress()` then passed to the `decode_session()` method (inherited from the superclass). If an exception is raised during `decode_session()` then the session will be deleted from the server and a new session will be created via the application object `new_session()` method.

`save_session``()`
This is called from the `Application` class `save_session()` method at the end of the request processing sequence. If the session save flag has been cleared via the `set_save_session()` method then the session is not saved.

Before saving a session the method calls the superclass `encode_session()` then calls `zlib.compress()` and `base64.encodestring()` to convert the session to plain text which is passed to the `put_session()` application method to save the session.

`remove_session``()`
This is called from the `Application` class `remove_session()` method. The method calls the superclass `remove_session()` then calls the `del_session()` application method to remove the session at the server.

10.7.5 SessionFileContextMixin

This class works in concert with the `SessionFileAppMixin` application mixin class to store session data in the local file-system. All management of session data storage is performed by the application class.

Inherits from the `SessionBase` class so you must call the constructor if you subclass this class.

`__init__``()`
: When you inherit from the `SessionFileContextMixin` class you must call this constructor.

`sesid``()`
: Returns the session id.

`load_session``()`
: This is usually called from the `Application` class `load_session()` method. Retrieves the session id and then either retrieves an existing session or creates a new session via the application object.

  If an existing session is retrieved it is passed to the `decode_session()` method (inherited from the superclass). If an exception is raised during `decode_session()` then the session will be deleted from the server and a new session will be created via the application object `new_session()` method.

`save_session``()`
: This is called from the `Application` class `save_session()` method at the end of the request processing sequence. If the session save flag has been cleared via the `set_save_session()` method then the session is not saved.

  Before saving a session the method calls the superclass `encode_session()` then calls the `put_session()` application method to save the session.

`remove_session``()`
: This is called from the `Application` class `remove_session()` method. The method calls the superclass `remove_session()` then calls the `del_session()` application method to remove the session at the server.

10.7.6 BranchingSessionMixin

A persistent problem with server-side sessions is the browser state getting out of synchronisation with the application state. This occurs when the browser “back” button is used, or when a form is reloaded (this is logically equivilent to a “back” then a resubmission of the old form state).

One solution to this problem is to maintain a server-side session for each interaction with the browser, rather than a single session per client that is recycled for each interaction. A unique session identifier is stored in a hidden form field, which allows us to retrieve the appropriate version of the session on form submission (the hidden field value is rolled back with the browser state when the “back” button is used, unlike a cookie). This class provides a drop-in replacement for the `SessionServerContextMixin` and implements this session-per-interaction behaviour.

`__init__``()`
: When you inherit from the `BranchingSessionMixin` class you must call this constructor.

`sesid``()`
: Returns the session id.

`load_session``()`
: This is usually called from the `Application` class `load_session()` method. Retrieves the session id and then either retrieves an existing session or creates a new session via the application object.

  If an existing session is retrieved it is passed to the `decode_session()` method (inherited from the superclass). If an exception is raised during `decode_session()` then the session will be deleted from the server and a new session will be created via the application object `new_session()` method.

`save_session``()`
: This is called from the `Application` class `save_session()` method at the end of the request processing sequence. If the session save flag has been cleared via the `set_save_session()` method then the session is not saved.

  Before saving a session the method calls the superclass `encode_session()` then calls the `put_session()` application method to save the session.

remove_session()
This is called from the `Application` class `remove_session()` method. The method calls the superclass `remove_session()` then calls the `del_session()` application method to remove the session at the server.

form_close()
Called just before the `<al-form>` tag is closed. If the session is flagged to be saved a hidden field named `__albsessid__` containing the session identifier is written to the output.

Note that this method is also present in the `RecorderMixin`, so if you inherit from the `BranchingSessionMixin` class you must define a `form_close()` method in the derived class which calls this method in both of the super classes.

10.8 SessionAppMixin Classes

10.8.1 SessionServerAppMixin

The application mixin works in concert with the `SessionServerContextMixin` execution context method to store sessions in the Albatross session server.

Whenever there are problems communicating with the session server the class raises a `SessionServerError` exception, which is a subclass of `SessionError`. Unless you have a reason to do otherwise, catch `SessionError` rather than `SessionServerError`, as this allows other Session classes to be substituted with minimal change.

__init__(*appid, [server ‘‘= ’localhost’‘‘], [port ‘‘= 34343‘‘], [age ‘‘= 1800‘‘]*)
When you inherit from the `SessionServerAppMixin` class you must call this constructor.

The *appid* argument specifies the name of the cookie attribute which is used to store the session id. This uniquely identifies the application at the web server. Multiple applications can share sessions by defining the same value in this argument.

The *server* and *port* arguments specify the location of the Albatross session server. By using a session server you can have a number of web serving machines which transparently share session data.

The *age* argument specifies how long (in seconds) an idle session will be stored at the server before it is discarded.

A connection to the session server is established. The connection will be kept open for the lifetime of the application object.

ses_appid()
Returns the *appid* argument which was passed to the constructor.

get_session(*sesid*)
Returns the session identified by *sesid* argument and the *appid* passed to the constructor. If no such session exists `None` is returned.

new_session()
Returns a new session id for the *appid* passed to the constructor.

put_session(*sesid, text*)
Saves the *text* argument as session data for the session identified by *sesid* argument and the *appid* passed to the constructor.

del_session(*sesid*)
Removes the session identified by *sesid* argument and the *appid* passed to the constructor.

10.8.2 SessionFileAppMixin

The application mixin works in concert with the `SessionFileContextMixin` execution context method to store sessions in the local file-system.

Whenever there are problems reading or writing sessions from or to disk, the class raises a `SessionFileError` exception, which is a subclass of `SessionError`. Unless you have a reason to do otherwise, catch `SessionError` rather than `SessionFileError`, as this allows other Session classes to be substituted with minimal change.

`__init__`(*appid, session_dir*)
When you inherit from the `SessionFileAppMixin` class you must call this constructor.

The *appid* argument specifies the name of the cookie attribute which is used to store the session id. This uniquely identifies the application at the web server. Multiple applications can share sessions by defining the same value in this argument.

The *session_dir* argument specifies the location on the local file-system in which this application's sessions will be stored.

The directory should not be publicly readable, as the session file names are the session id's (knowing a session id allows an attacker to steal that session).

Sessions recorded via this class are not automatically aged. An external process will be required to clean orphaned sessions from the session directory (for example, by removing any file that has not been accessed in the last two hours).

`ses_appid`()
Returns the *appid* argument which was passed to the constructor.

`get_session`(*sesid*)
Returns the session identified by *sesid* argument and the *appid* passed to the constructor. If no such session exists `None` is returned.

`new_session`()
Returns a new session id for the *appid* passed to the constructor.

Note that if the *session_dir* passed to the constructor does not already exist, this method will attempt to create it.

`put_session`(*sesid, text*)
Saves the *text* argument as session data for the session identified by *sesid* argument and the *appid* passed to the constructor.

`del_session`(*sesid*)
Removes the session identified by *sesid* argument and the *appid* passed to the constructor.

10.9 PickleSignMixin Classes

This is mixed with the application class to sign or modify pickles before sending them to the browser and to undo and check that modification on the return trip. When processing modified pickles returned from the browser the class discards pickles which do not pass the security check.

There is only one mixin supplied for this function; the `PickleSignMixin` class. Pickle strings are combined with the secret string which was passed to the application constructor as the *secret* argument using the HMAC-SHA1 algorithm. The resulting signature is then prepended to the pickle. On the return trip the HMAC-SHA1 sign is compared with the result of the signing process on the pickle returned from the browser. If the two signs are not the same, the pickle is discarded.

The process does not prevent users from seeing the contents of a pickle, rather it provides an assurance of its authenticity.

The mixin has the following interface.

`__init__`(*secret*)
When you inherit from the `PickleSignMixin` class you must call this constructor.

The *secret* argument is the secret key which is combined with the pickle to produces the HMAC-SHA1 signature.

pickle_sign(*text*)
Generates an HMAC-SHA1 signed copy of the *text* argument, using the *secret* constructor argument as key.

pickle_unsign(*text*)
Compares the HMAC-SHA1 signature on the given *text*, and if valid, returns the unsigned text. If the signature does not match, a `SecurityError` exception is raised.

10.10 PageMixin Classes

The choice of mixin for this functionality determines how Albatross will locate the code to process the browser request and display the response for each page. Albatross supplies three classes; `PageModuleMixin`, `RandomPageModuleMixin`, and `PageObjectMixin`.

10.10.1 PageModuleMixin

This class uses a separate Python module for each page in the application. This scales very well at runtime because at most two modules will be loaded for each request; one to process the browser request, and possibly another to display the response. Page modules are cached for further efficiency. The class is designed to be used where the application controls the sequence of pages seen in a browser session so the start page is also specified in the constructor.

Application page modules are loaded from a base directory which is specified by the constructor *base_dir* argument. The current application page is identified by the path to the page module relative to the module base directory. Page identifiers consist of an optional path component followed by a module name without extension. For example `"login"`, `"user/list"`, `"home-page/default"`.

Page modules are loaded into a dummy `__alpage__` namespace to avoid conflicts with python modules, so loading `"user/list"` actually imports the module as `__alpage__.user.list`.

To support pickling of instances defined in a page module, a dummy hierarchy of modules needs to be created. In the `__alpage__.user.list` case mentioned above, a temporary dummy module called `__alpage__.user` is registered. This will be replaced by the real `user` module later if it is loaded.

Note also that the `SessionBase` mixin uses an import hook while decoding the session to redirects attempts to load page modules (those that being with `__alpage__`) to the `load_page_module()` method.

Page modules handled by this mixin have the following interface:

page_enter(*ctx, [...]*)
If this function is present in the new page module it will be called whenever your application code calls the execution context `set_page()` method. For application types which define a start page this method is called in the start page when a new session is created.

The *ctx* argument contains the execution context. Any extra arguments which are passed to the `set_page()` method are passed as optional extra arguments to this function.

page_leave(*ctx*)
If this function is present in the current page module it will be called whenever your application code changes to another page by calling the execution context `set_page()` method.

The *ctx* argument contains the execution context.

page_process(*ctx*)
If this function is present in the page module it will be called when the application object executes the `process_request()` method. This occurs if the browser request was successfully validated.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

page_display(*ctx*)
This is the only mandatory page module function. The application object calls this function when it executes the `display_response()` method as the final step before saving the browser session.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

The `PageModuleMixin` class has the following interface.

`__init__`(*base_dir, start_page*)
When you inherit from the `PageModuleMixin` class you must call this constructor.

The *base_dir* argument specifies the path of the root directory where page modules are loaded from. When deploying your application as a CGI program you can specify a relative path from the location of the application mainline. Apache sets the current directory to root so when using `mod_python` deployment you will need to specify a path relative to the root.

The *start_page* argument is a page identifier which specifies the first page that new browser session will see.

`module_path`()
Returns the *base_dir* which was passed to the constructor.

`start_page`()
Returns the *start_page* which was passed to the constructor.

`load_page`(*ctx*)
This method implements part of the standard application processing sequence. It is called immediately after restoring the browser session. The *ctx* argument is the execution context for the current browser request.

If no current page is defined in *ctx* then the method will invoke `ctx.set_page()` passing the page specified as the *start_page* argument to the application constructor.

The actual page module load is performed via the `load_page_module()` method.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

`load_page_module`(*ctx, name*)
Loads the page module identified by the *name* argument and saves a reference to the module in the `page` member.

`page_enter`(*ctx, args*)
Called when your application code calls the execution context `set_page()` method. The *ctx* argument is the execution context for the current browser request. The *args* argument is a tuple which contains all optional extra arguments which were passed to the `set_page()` method.

The page module `page_enter()` function is called by this method.

`page_leave`(*ctx*)
Called before changing pages when your application code calls the execution context `set_page()` method. The *ctx* argument is the execution context for the current browser request.

The page module `page_leave()` function is called by this method.

`process_request`(*ctx*)
This method implements part of the standard application processing sequence. It is called if the browser request is successfully validated. The *ctx* argument is the execution context for the current browser request.

The page module `page_process()` function is called by this method.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

`display_response`(*ctx*)
This method implements part of the standard application processing sequence. It is called as the final stage just before the session is saved. The *ctx* argument is the execution context for the current browser request.

The page module `page_display()` function is called by this method.

Refer to section the *Albatross Application Model* section for an overview of the application processing sequence.

10.10.2 RandomPageModuleMixin

This class inherits from the `PageModuleMixin` class. It redefines the way in which page modules are selected.

Instead of the application calling the `set_page()` execution context method, the URL in the browser request controls which page module is loaded and processed for each request.

Page module management is inherited from `PageModuleMixin`. The *base_dir* argument to the constructor determines the root directory where modules are loaded from.

Page modules handled by this mixin have the following interface:

`page_enter`(*ctx*)
: If this function is present in the page module it will be called every time the page module is used for a browser request.

  The *ctx* argument contains the execution context.

`page_process`(*ctx*)
: If this function is present in the page module it will be called when the application object executes the `process_request()` method. This occurs if the browser request was successfully validated.

  Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

`page_display`(*ctx*)
: This is the only mandatory page module function. The application object calls this function when it executes the `display_response()` method as the final step before saving the browser session.

  Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

The `RandomPageModuleMixin` class has the following interface.

`load_page`(*ctx*)
: This method implements part of the standard application processing sequence. It is called immediately after restoring the browser session. The *ctx* argument is the execution context for the current browser request.

  The `get_page_from_uri()` method is called to determine the identifier of the page module that will be loaded. The identifier is then passed to the `load_page_module()` method (which is inherited from `PageModuleMixin`).

  Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

`get_page_from_uri`(*ctx, uri*)
: The method uses the `urlparse()` function from the standard Python `urlparse()` module to extract the path component from both the *uri* parameter and the value returned by the `base_url()` method (which returns the *base_url* argument to the application constructor).

  The *path* component of the *base_url* is then used to split the *path* component of the *uri*. Element one (first split to the right of *base_url*) of the resulting string list is returned as the page identifier.

  Override this method in your application if you wish to implement a your own scheme for mapping the request onto a page identifier.

`load_badurl_template`(*ctx*)
: Called when your page template identified by the request URL does not exist. The *ctx* argument is the execution context for the current browser request.

  Override this method if you want to supply a different error page template.

`page_enter`(*ctx*)
: Called as soon as the page module has been loaded. The *ctx* argument is the execution context for the current browser request.

  The page module `page_enter()` function is called by this method if a page module was located by the `load_page()` method.

`process_request`(*ctx*)
: This method implements part of the standard application processing sequence. It is called if the browser request is successfully validated. The *ctx* argument is the execution context for the current browser request.

  The page module `page_process()` function is called by this method if a page module was located by the `load_page()` method.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

`display_response`(*ctx*)
This method implements part of the standard application processing sequence. It is called as the final stage just before the session is saved. The *ctx* argument is the execution context for the current browser request.

The page module `page_display()` function is called by this method if a page module was located by the `load_page()` method.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

10.10.3 PageObjectMixin

This class is intended for applications which do not require a separate Python module for each page in the application. Page processing is performed by a set of objects which the application registers with this class. The class is designed to be used where the application controls the sequence of pages seen in a browser session so the start page is specified in the constructor.

Application page objects must be registered before they can be used. Typically you will register the page objects immediately after constructing your application object. Since the current application page is identified by an internal value, any hashable pickleable value can be used as an identifier.

Page objects handled by this mixin have the following interface:

`page_enter`(*ctx, [...]*)
If this method is present in the new page object it will be called whenever your application code changes current the page by calling the execution context `set_page()` method. For application types which define a start page this method is called in the start page when a new session is created.

The *ctx* argument contains the execution context. Any extra arguments which are passed to the `set_page()` method are passed as optional extra arguments to this method.

`page_leave`(*ctx, [...]*)
If this method is present in the old page object it will be called whenever your application code changes current the page by calling the execution context `set_page()` method.

The *ctx* argument contains the execution context.

`page_process`(*ctx*)
If this method is present in the page object it will be called when the application object executes the `process_request()` method. This occurs if the browser request was successfully validated.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

`page_display`(*ctx*)
This is the only mandatory page object function. The application object calls this method when it executes the `display_response()` method as the final step before saving the browser session.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

The `PageObjectMixin` class has the following interface.

`__init__`(*start_page*)
When you inherit from the `PageObjectMixin` class you must call this constructor.

The *start_page* argument is a page identifier which specifies the first page that new browser session will see.

`module_path`()
Returns `None`.

`start_page`()
Returns the *start_page* argument which was passed to the constructor.

`register_page`(*name, obj*)
You must call this method to register every page object in your application. The *name* argument defines the page identifier which is used to select the page object specified in the *obj* argument. All pages must be registered before they can be used.

load_page(*ctx*)
This method implements part of the standard application processing sequence. It is called immediately after restoring the browser session. The *ctx* argument is the execution context for the current browser request.

If no current page is defined in *ctx* then the method will invoke `ctx.set_page()` passing the page specified as the *start_page* argument to the application constructor.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

page_enter(*ctx, args*)
Called when your application code calls the execution context `set_page()` method. The *ctx* argument is the execution context for the current browser request. The *args* argument is a tuple which contains all optional extra arguments which were passed to the `set_page()` method.

The page object `page_enter()` method is called by this method.

page_leave(*ctx*)
Called before changing pages when your application code calls the execution context `set_page()` method. The *ctx* argument is the execution context for the current browser request.

The page object `page_leave()` method is called by this method.

process_request(*ctx*)
This method implements part of the standard application processing sequence. It is called if the browser request is successfully validated. The *ctx* argument is the execution context for the current browser request.

The page object `page_process()` method is called by this method.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

display_response(*ctx*)
This method implements part of the standard application processing sequence. It is called as the final stage just before the session is saved. The *ctx* argument is the execution context for the current browser request.

The page object `page_display()` method is called by this method.

Refer to the *Albatross Application Model* section for an overview of the application processing sequence.

10.11 Request Classes

The choice of `Request` class determines how you wish to deploy your application. Albatross supplies a number of pre-built Request implementations suited to various deployment methods. These include:

Deployment Method	Request Module
CGI	`albatross.cgiapp`
mod_python	`albatross.apacheapp`
FastCGI_python	`albatross.fcgiapp`
Stand-alone Python HTTP server	`albatross.httpdapp`

You can also develop your own `Request` class to deploy an Albatross application in other ways.

All `Request` classes implement the same interface. Much of this interface can be supplied by the `RequestBase` mixin.

has_field(*name*)
Returns `TRUE` if the field identified by the *name* argument is present in the request.

field_value(*name*)
Return the value of the field identified by the *name* argument.

field_file(*name*)
Returns an object that contains the value of a file input field.

field_names()
Return a list of all all fields names in the request.

get_uri()
: Return the URL which the browser used to perform the request.

get_servername()
: Return the name of the server (Apache ServerName setting).

get_header(*name*)
: Return the value of the HTTP header identified in the *name* argument.

write_header(*name, value*)
: Add a header named *name* with the value *value* to the response. This method should not be called once you have started sending content to the browser.

end_headers()
: Signal to the Request object that header generation has finished and that you are ready to start sending content.

redirect(*loc*)
: Send a `"301 Moved Permanently"` response back to the browser.

write_content(*data*)
: Send *data* as part of the request response.

set_status(*status*)
: Sets the HTTP status code of the response. Defaults to 200. For deployment methods based on the cgiapp module, this value is used to derive the `Status:` header. The apacheapp module uses it to set the `status` member of the mod_python `request` object.

status(*num*)
: Return the saved value for the HTTP status code.

return_code()
: Returns a value which should be returned from the `Application` class `run()` method. For most deployment methods, this is `None`, however the `mod_python` requires that `mod_python.apache.OK` be returned if application emits any content. Your `mod_python` application should include code such as this:

```
from albatross.apacheapp import Request
   :
   :
def handler(req):
    return app.run(Request(req))
```

CHAPTER

ELEVEN

PREPACKAGED APPLICATION AND EXECUTION CONTEXT CLASSES

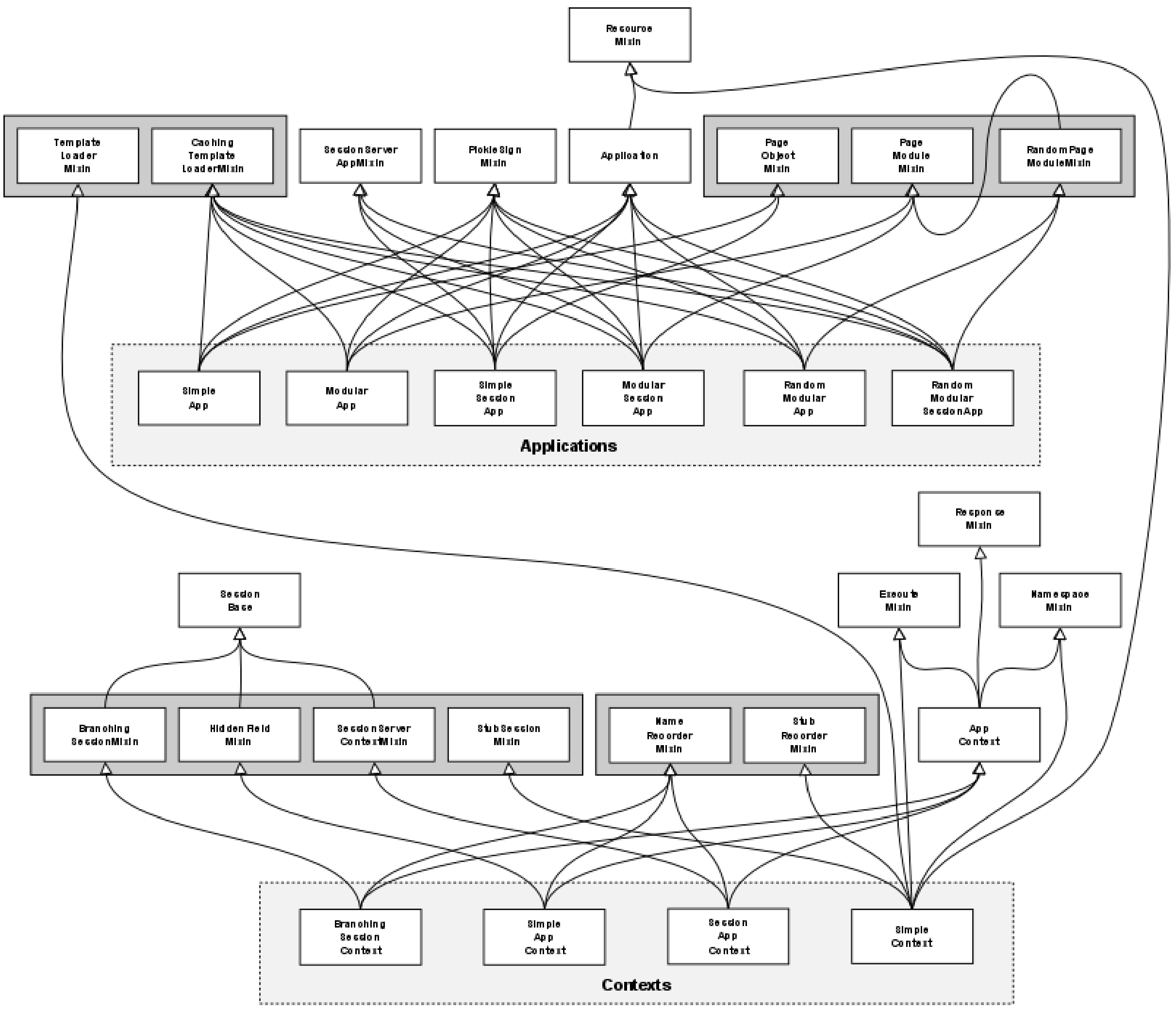


Figure 11.1: Albatross Classes

11.1 The `SimpleContext` Execution Context

The `SimpleContext` class is provided for applications which only make use of the Albatross template functionality. If you look at the implementation of the class you will note that it is constructed from a number of mixin classes. Each of these classes implements some of the functionality required for interpreting Albatross templates.

Diagrammatically the `SimpleContext` class looks like this:

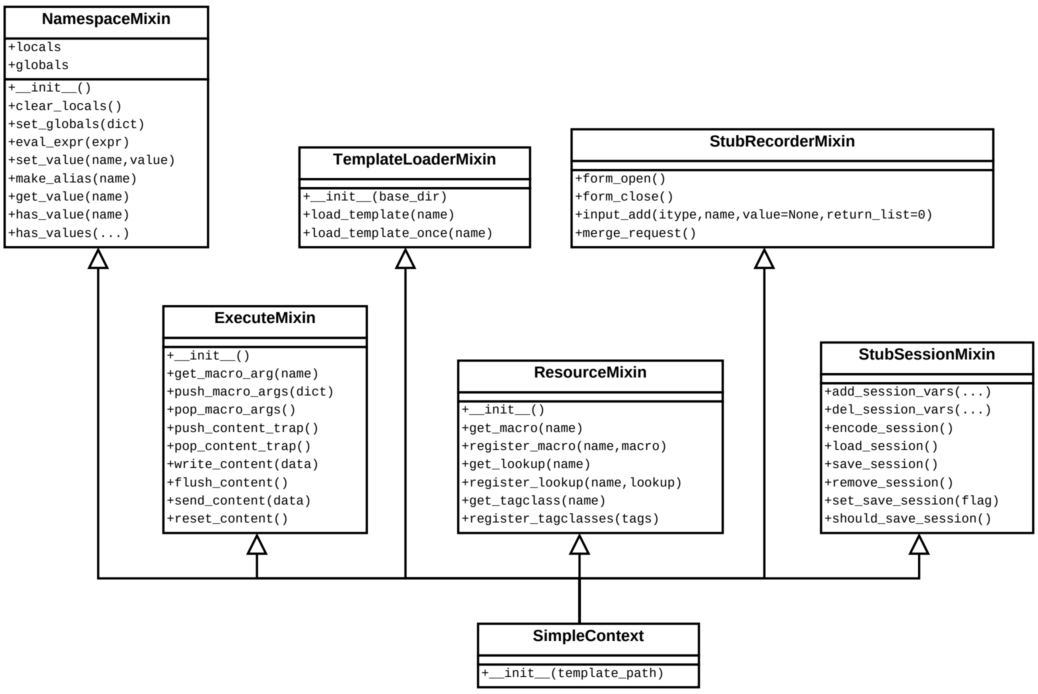


Figure 11.2: The `SimpleContext` class

By implementing the execution context semantics in a collection of mixin classes Albatross allows you to change semantics by substituting mixins which implement the same interface. This is very useful when using the Albatross application objects.

> `NamespaceMixin` This mixin provides a local namespace for evaluating the expressions embedded in the `expr` tag attributes. Application code places values into the `locals` member to make them available for display by the template files.
>
> You will probably always use this mixin in your execution context.
>
> `ExecuteMixin` This mixin provides a sort of virtual machine which is required by the template file interpreter. It maintains a macro argument stack for expanding macros, and it is used to accumulate HTML produced by template execution.
>
> You will probably always use this mixin in your execution context.
>
> `ResourceMixin` This mixin provides a registry of template resources which only need to be defined once. Specifically the class provides a dictionary of Python classes which implement template file tags, a dictionary of template macros, and a dictionary of template lookup tables.
>
> If you are using Albatross application functionality, you will almost certainly use this mixin in your application class, not the execution context.
>
> `TemplateLoaderMixin` This mixin is a very simple template file loader. You will almost certainly use the `CachingTemplateLoaderMixin` in your application object instead of this mixin when you use the Albatross application objects.
>
> `StubRecorderMixin` Albatross provides special versions of the standard HTML `<input>`, `<select>`, and `<form>` tags. As these tags are converted to HTML they report back to the execution context. Applications which do not need to record the contents of each form can use this mixin to ignore the form record.
>
> `StubSessionMixin`

Albatross provides an application session. Applications which do not a session can use this mixin to disable session functionality.

Collectively these classes provide all of the functionality which is required to execute Albatross templates. The following table contains a list of all methods defined in the context.

Method	Mixin
`add_session_vars(*names)()`	`StubSessionMixin`
`clear_active_select()`	`ExecuteMixin`
`clear_locals()`	`NamespaceMixin`
`del_session_vars(*names)()`	`StubSessionMixin`
`discard_file_resources(filename)()`	`ResourceMixin`
`encode_session()`	`StubSessionMixin`
`eval_expr(expr)()`	`NamespaceMixin`
`flush_content()`	`ExecuteMixin`
`flush_html()`	`ExecuteMixin`
`form_close()`	`StubRecorderMixin`
`form_open()`	`StubRecorderMixin`
`get_active_select()`	`ExecuteMixin`
`get_lookup(name)()`	`ResourceMixin`
`get_macro(name)()`	`ResourceMixin`
`get_macro_arg(name)()`	`ExecuteMixin`
`get_tagclass(name)()`	`ResourceMixin`
`get_value(name)()`	`NamespaceMixin`
`has_value(name)()`	`NamespaceMixin`
`has_values(*names)()`	`NamespaceMixin`
`input_add(itype, name, value, return_list)()`	`StubRecorderMixin`
`load_session()`	`StubSessionMixin`
`load_template(name)()`	`TemplateLoaderMixin`
`load_template_once(name)()`	`TemplateLoaderMixin`
`make_alias(name)()`	`NamespaceMixin`
`merge_request()`	`StubRecorderMixin`
`merge_vars(*vars)()`	`NamespaceMixin`
`pop_content_trap()`	`ExecuteMixin`
`pop_macro_args()`	`ExecuteMixin`
`push_content_trap()`	`ExecuteMixin`
`push_macro_args(args, defaults)()`	`ExecuteMixin`
`register_lookup(name, lookup)()`	`ResourceMixin`
`register_macro(name, macro)()`	`ResourceMixin`
`register_tagclasses(*tags)()`	`ResourceMixin`
`remove_session()`	`StubSessionMixin`
`reset_content()`	`ExecuteMixin`
`save_session()`	`StubSessionMixin`
`send_content(data)()`	`ExecuteMixin`
`set_active_select(select, value)()`	`ExecuteMixin`
`set_globals(dict)()`	`NamespaceMixin`
`set_save_session(flag)()`	`StubSessionMixin`
`set_value(name, value)()`	`NamespaceMixin`
`should_save_session()`	`StubSessionMixin`
`write_content(data)()`	`ExecuteMixin`

Looking inside the `context` module you will notice some mixin classes which provide alternatives for some of the context functionality. The `CachingTemplateLoaderMixin` class can be used to replace the `TemplateLoaderMixin`. Likewise the `NameRecorderMixin` class is a drop-in replacement for the `StubRecorderMixin` class. These alternatives are used by some of the prepackaged application objects.

Although all of the template file examples in the Templates User Guide used the `SimpleContext` class as the execution context, you are much more likely to use something derived from the `AppContext` class defined in the app module. Since Albatross creates a new execution context to process each browser request, it makes sense to manage tag classes, macros, and lookup tables somewhere other than in the execution context.

11.2 The `AppContext` Base Class

All execution contexts used by Albatross application classes are derived from the `AppContext` class. The class acts as a proxy and redirects all `ResourceMixin` and `TemplateLoader` class methods to the application object. This allows the application to cache resources so that they do not need to be defined for every request.

The class places very few assumptions about how you wish to structure and deploy your application and is suitable as a base class for all application execution context classes.

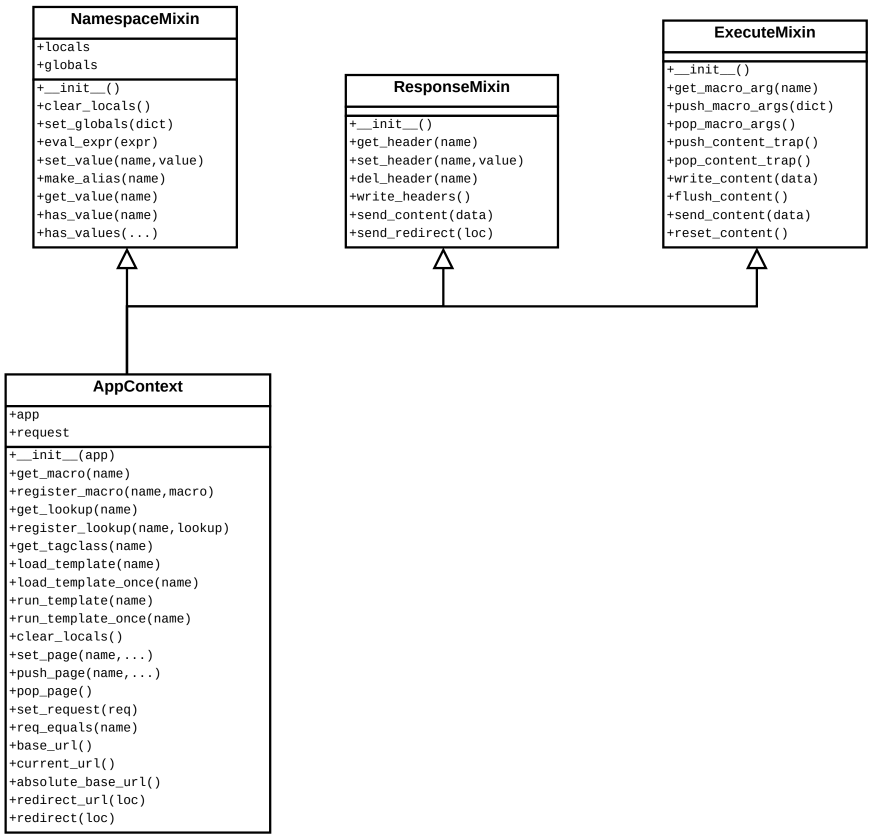


Figure 11.3: The `AppContext` class

The methods available in `AppContext` and the location of their definition are show below.

Method	Mixin
`absolute_base_url()`	`AppContext`
`add_header(name, value)()`	`ResponseMixin`
`base_url()`	`AppContext`
`clear_active_select()`	`ExecuteMixin`
`clear_locals()`	`AppContext`
`current_url()`	`AppContext`
`del_header(name)()`	`ResponseMixin`
`eval_expr(expr)()`	`NamespaceMixin`
`flush_content()`	`ExecuteMixin`
`flush_html()`	`ExecuteMixin`
`get_active_select()`	`ExecuteMixin`
`get_header(name)()`	`ResponseMixin`

Continued on next page

Table 11.2 – continued from previous page

`get_lookup(name)()`	AppContext
`get_macro(name)()`	AppContext
`get_macro_arg(name)()`	ExecuteMixin
`get_tagclass(name)()`	AppContext
`get_value(name)()`	NamespaceMixin
`has_value(name)()`	NamespaceMixin
`has_values(*names)()`	NamespaceMixin
`load_template(name)()`	AppContext
`load_template_once(name)()`	AppContext
`make_alias(name)()`	NamespaceMixin
`merge_vars(*vars)()`	NamespaceMixin
`parsed_request_uri()`	AppContext
`pop_content_trap()`	ExecuteMixin
`pop_macro_args()`	ExecuteMixin
`pop_page(target_page)()`	AppContext
`push_content_trap()`	ExecuteMixin
`push_macro_args(args, defaults)()`	ExecuteMixin
`push_page(name, *args)()`	AppContext
`redirect(loc)()`	AppContext
`redirect_url(loc)()`	AppContext
`register_lookup(name, lookup)()`	AppContext
`register_macro(name, macro)()`	AppContext
`req_equals(name)()`	AppContext
`reset_content()`	ExecuteMixin
`run_template(name)()`	AppContext
`run_template_once(name)()`	AppContext
`send_content(data)()`	ResponseMixin
`send_redirect(loc)()`	ResponseMixin
`set_active_select(select, value)()`	ExecuteMixin
`set_globals(dict)()`	NamespaceMixin
`set_header(name, value)()`	ResponseMixin
`set_page(name, *args)()`	AppContext
`set_request(req)()`	AppContext
`set_value(name, value)()`	NamespaceMixin
`write_content(data)()`	ExecuteMixin
`write_headers()`	ResponseMixin

There are a number of new methods and attributes introduced by the class.

`__init__`(*app*)
When you inherit from the `AppContext` class you must call this constructor.

The *app* argument specifies the application object. This is saved in the `app` member.

`app`
Stores the *app* argument to the constructor.

`get_macro`(*name*)
Returns the result of the `get_macro()` method of the application in the `app` member.

`register_macro`(*name, macro*)
Returns the result of the `register_macro()` method of the application in the `app` member.

`get_lookup`(*name*)
Returns the result of the `get_lookup()` method of the application in the `app` member.

`register_lookup`(*name, lookup*)
Returns the result of the `register_lookup()` method of the application in the `app` member.

`get_tagclass`(*name*)
Returns the result of the `get_tagclass()` method of the application in the `app` member.

load_template(*name*)
Returns the result of the `load_template()` method of the application in the `app` member.

load_template_once(*name*)
Returns the result of the `load_template_once()` method of the application in the `app` member.

run_template(*name*)
Calls the application `load_template()` method to load the template specified in the *name* argument and sets the execution context global namespace to the globals of the function which called this method. The template `to_html()` method is then called to execute the template.

run_template_once(*name*)
Calls the application `load_template_once()` method. If the template specified in the *name* argument is loaded or reloaded the method sets the execution context global namespace to the globals of the function which called this method, then the template `to_html()` method is then called to execute the template.

clear_locals()
Overrides the `NamespaceMixin` `clear_locals()` method to retain the `__page__` local namespace value.

set_page(*name, [...]*)
Sets the current application page to that specified in the *name* argument. If changing pages and there is a current page defined then before changing pages the `page_leave()` function/method will be called for the current page. The `locals.__page__` member is then set to *name* and the new page is loaded. Any addition arguments passed to the method will be passed to the `page_enter()` function/method code which is associated with the new page.

Refer to *PageMixin Classes* for an explanation of page functions/methods.

push_page(*name, [...]*)
Pushes an application page onto the page stack. The current page can be returned to by calling the `pop_page()` method. The `page_leave()` function/method of the current page is not called. The new page is loaded and it's `page_enter()` function/method is called. Any additional arguments given will be passed to the `page_enter()` function/method associated with the new page.

pop_page()
Pops the current page from the page stack and returns to the page that was current when the `push_page()` method was called. The `page_leave()` function/method of the current page is called prior to loading the original page. The `page_enter()` function/method of the original page is not called.

set_request(*req*)
Saves the browser request specified in the *req* argument as the `request`.

req_equals(*name*)
Returns whether or not the browser request contains a non-empty field with a name which matches the *name* argument.

base_url()
Returns the result of the application `base_url()` method.

current_url()
Returns the path component (see the standard `urlparse` module) of the URI used to request the current page.

absolute_base_url()
Returns the *base_url* parameter to the application constructor transformed into an absolute URL.

redirect_url(*loc*)
Returns an absolute URL for the application page identifier specified in the *loc* parameter.

redirect(*loc*)
Raises a `Redirect` exception requesting a redirect to the location in the *loc* parameter from the application `run()` method.

If the *loc* parameter contains either a scheme or netloc (from the standard `urlparse` module), or begins with a "/" then is it used without modification for the `Redirect` exception. Other forms of *loc* are assumed to be page identifiers and are passed to `redirect_url()` before being raised as a `Redirect` exception.

11.3 Context classes:

11.3.1 The `SimpleAppContext` Class

The `SimpleAppContext` class is intended to be used for applications which store state at the browser in hidden HTML fields. An inheritance diagram illustrates the relationship to the `SimpleContext` class described above.

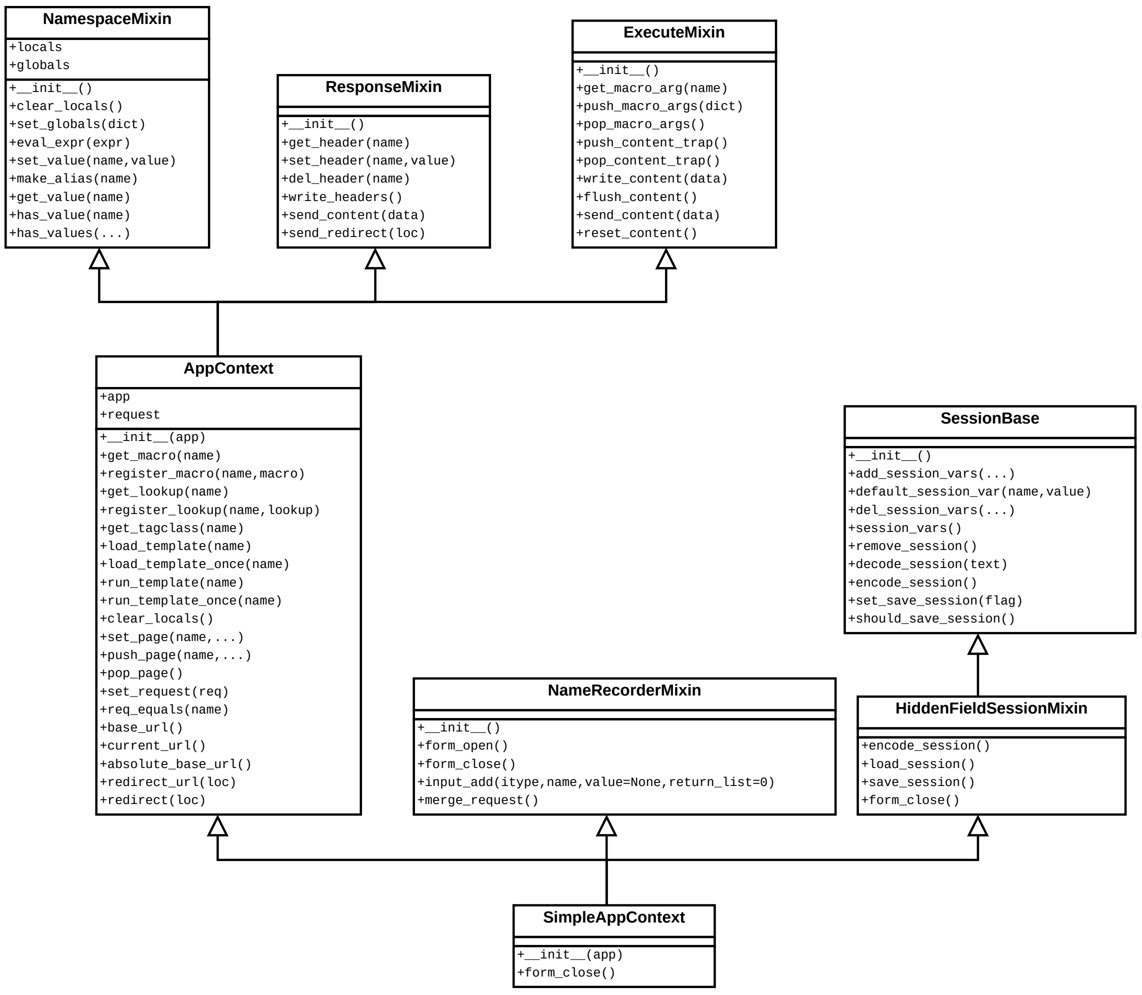


Figure 11.4: The `SimpleAppContext` class

The methods available in `SimpleAppContext` and the location of their definition are show below.

Method	Mixin
`absolute_base_url()`	`AppContext`
`add_header(name, value)()`	`ResponseMixin`
`add_session_vars(*names)()`	`SessionBase`
`base_url()`	`AppContext`
`clear_active_select()`	`ExecuteMixin`
`clear_locals()`	`AppContext`
`current_url()`	`AppContext`
`decode_session(text)()`	`SessionBase`

Continued on next page

Table 11.3 – continued from previous page

`default_session_var(name, value)()`	`SessionBase`
`del_header(name)()`	`ResponseMixin`
`del_session_vars(*names)()`	`SessionBase`
`encode_session()`	`HiddenFieldSessionMixin`
`eval_expr(expr)()`	`NamespaceMixin`
`flush_content()`	`ExecuteMixin`
`flush_html()`	`ExecuteMixin`
`form_close()`	`SimpleAppContext`
`form_open()`	`NameRecorderMixin`
`get_active_select()`	`ExecuteMixin`
`get_header(name)()`	`ResponseMixin`
`get_lookup(name)()`	`AppContext`
`get_macro(name)()`	`AppContext`
`get_macro_arg(name)()`	`ExecuteMixin`
`get_tagclass(name)()`	`AppContext`
`get_value(name)()`	`NamespaceMixin`
`has_value(name)()`	`NamespaceMixin`
`has_values(*names)()`	`NamespaceMixin`
`input_add(itype, name, unused_value, return_list)()`	`NameRecorderMixin`
`load_session()`	`HiddenFieldSessionMixin`
`load_template(name)()`	`AppContext`
`load_template_once(name)()`	`AppContext`
`make_alias(name)()`	`NamespaceMixin`
`merge_request()`	`NameRecorderMixin`
`merge_vars(*vars)()`	`NamespaceMixin`
`parsed_request_uri()`	`AppContext`
`pop_content_trap()`	`ExecuteMixin`
`pop_macro_args()`	`ExecuteMixin`
`pop_page(target_page)()`	`AppContext`
`push_content_trap()`	`ExecuteMixin`
`push_macro_args(args, defaults)()`	`ExecuteMixin`
`push_page(name, *args)()`	`AppContext`
`redirect(loc)()`	`AppContext`
`redirect_url(loc)()`	`AppContext`
`register_lookup(name, lookup)()`	`AppContext`
`register_macro(name, macro)()`	`AppContext`
`remove_session()`	`SessionBase`
`req_equals(name)()`	`AppContext`
`reset_content()`	`ExecuteMixin`
`run_template(name)()`	`AppContext`
`run_template_once(name)()`	`AppContext`
`save_session()`	`HiddenFieldSessionMixin`
`send_content(data)()`	`ResponseMixin`
`send_redirect(loc)()`	`ResponseMixin`
`session_vars()`	`SessionBase`
`set_active_select(select, value)()`	`ExecuteMixin`
`set_globals(dict)()`	`NamespaceMixin`
`set_header(name, value)()`	`ResponseMixin`
`set_page(name, *args)()`	`AppContext`
`set_request(req)()`	`AppContext`
`set_save_session(flag)()`	`SessionBase`
`set_value(name, value)()`	`NamespaceMixin`
`should_save_session()`	`SessionBase`
`write_content(data)()`	`ExecuteMixin`
`write_headers()`	`ResponseMixin`

The `SimpleAppContext` class provides the following functionality to your application.

- Application state is stored inside hidden fields in the HTML.

  This function is performed by the `HiddenFieldSessionMixin` mixin class which places a hidden field named `__albstate__` inside each HTML form constructed using `<al-form>` tags. The session data is pickled, compressed, then base64 encoded. No encryption is performed, so this is not suitable for storing sensitive data.

  If you refer back to the Albatross application processing sequence described in the *Albatross Application Model* section, you will note where the session is loaded into and saved from the context. These steps correspond to the `load_session()` and `save_session()` methods of the execution context respectively.

  In the `HiddenFieldSessionMixin` class, the `load_session()` method retrieves the encoded session data from the `__albstate__` field in the browser request. The `save_session()` method does not do anything because the session has already been saved into each form produced in the HTML output.

- All input fields in an HTML form which are left empty by the browser will be set to `None` when the request is merged into the local namespace.

  The `NameRecorderMixin` mixin class encodes the names of all `<al-input>` fields in the form inside a hidden field named `__albform__`.

  Any input field which is left empty in the browser when the form is submitted will not exist in the browser request to the server. When the toolkit merges the browser request into the application context, the `__albform__` field is used to detect the fields missing from the browser request.

The methods implemented in the `SimpleAppContext` class are:

`__init__`(*app*)
  When you inherit from the `SimpleAppContext` class you must call this constructor.

  The *app* argument is passed to the `AppContext` constructor.

`form_close`()
  Invokes the `form_close()` method of the `NameRecorderMixin` class and `encode_session()` of the `HiddenFieldSessionMixin` class.

11.3.2 The `SessionAppContext` Class

The `SessionAppContext` class is intended to be used for applications which store state at the server. An inheritance diagram illustrates the relationship to the `SimpleAppContext` class described above.

The methods available in `SessionAppContext` and the location of their definition are show below.

Method	Mixin
`absolute_base_url()`	`AppContext`
`add_header(name, value)()`	`ResponseMixin`
`add_session_vars(*names)()`	`SessionBase`
`base_url()`	`AppContext`
`clear_active_select()`	`ExecuteMixin`
`clear_locals()`	`AppContext`
`current_url()`	`AppContext`
`decode_session(text)()`	`SessionBase`
`default_session_var(name, value)()`	`SessionBase`
`del_header(name)()`	`ResponseMixin`
`del_session_vars(*names)()`	`SessionBase`
`encode_session()`	`SessionBase`
`eval_expr(expr)()`	`NamespaceMixin`
`flush_content()`	`ExecuteMixin`
`flush_html()`	`ExecuteMixin`
`form_close()`	`NameRecorderMixin`
`form_open()`	`NameRecorderMixin`
`get_active_select()`	`ExecuteMixin`

Continued on next page

Table 11.4 – continued from previous page

get_header(name)()	ResponseMixin
get_lookup(name)()	AppContext
get_macro(name)()	AppContext
get_macro_arg(name)()	ExecuteMixin
get_tagclass(name)()	AppContext
get_value(name)()	NamespaceMixin
has_value(name)()	NamespaceMixin
has_values(*names)()	NamespaceMixin
input_add(itype, name, unused_value, return_list)()	NameRecorderMixin
load_session()	SessionServerContextMixin
load_template(name)()	AppContext
load_template_once(name)()	AppContext
make_alias(name)()	NamespaceMixin
merge_request()	NameRecorderMixin
merge_vars(*vars)()	NamespaceMixin
new_session()	SessionServerContextMixin
parsed_request_uri()	AppContext
pop_content_trap()	ExecuteMixin
pop_macro_args()	ExecuteMixin
pop_page(target_page)()	AppContext
push_content_trap()	ExecuteMixin
push_macro_args(args, defaults)()	ExecuteMixin
push_page(name, *args)()	AppContext
redirect(loc)()	AppContext
redirect_url(loc)()	AppContext
register_lookup(name, lookup)()	AppContext
register_macro(name, macro)()	AppContext
remove_session()	SessionServerContextMixin
req_equals(name)()	AppContext
reset_content()	ExecuteMixin
run_template(name)()	AppContext
run_template_once(name)()	AppContext
save_session()	SessionServerContextMixin
send_content(data)()	ResponseMixin
send_redirect(loc)()	ResponseMixin
sesid()	SessionServerContextMixin
session_vars()	SessionBase
set_active_select(select, value)()	ExecuteMixin
set_globals(dict)()	NamespaceMixin
set_header(name, value)()	ResponseMixin
set_page(name, *args)()	AppContext
set_request(req)()	AppContext
set_save_session(flag)()	SessionBase
set_value(name, value)()	NamespaceMixin
should_save_session()	SessionBase
write_content(data)()	ExecuteMixin
write_headers()	ResponseMixin

Externally the execution context is almost identical to that of the `SimpleAppContext` class. Instead of saving session data in hidden HTML fields, session data is loaded and saved via a session server which is managed by the application.

The class defines a number of extra methods.

`__init__`(*app*)

When you inherit from the `SessionAppContext` class you must call this constructor.

The *app* argument is passed to the `AppContext` constructor.

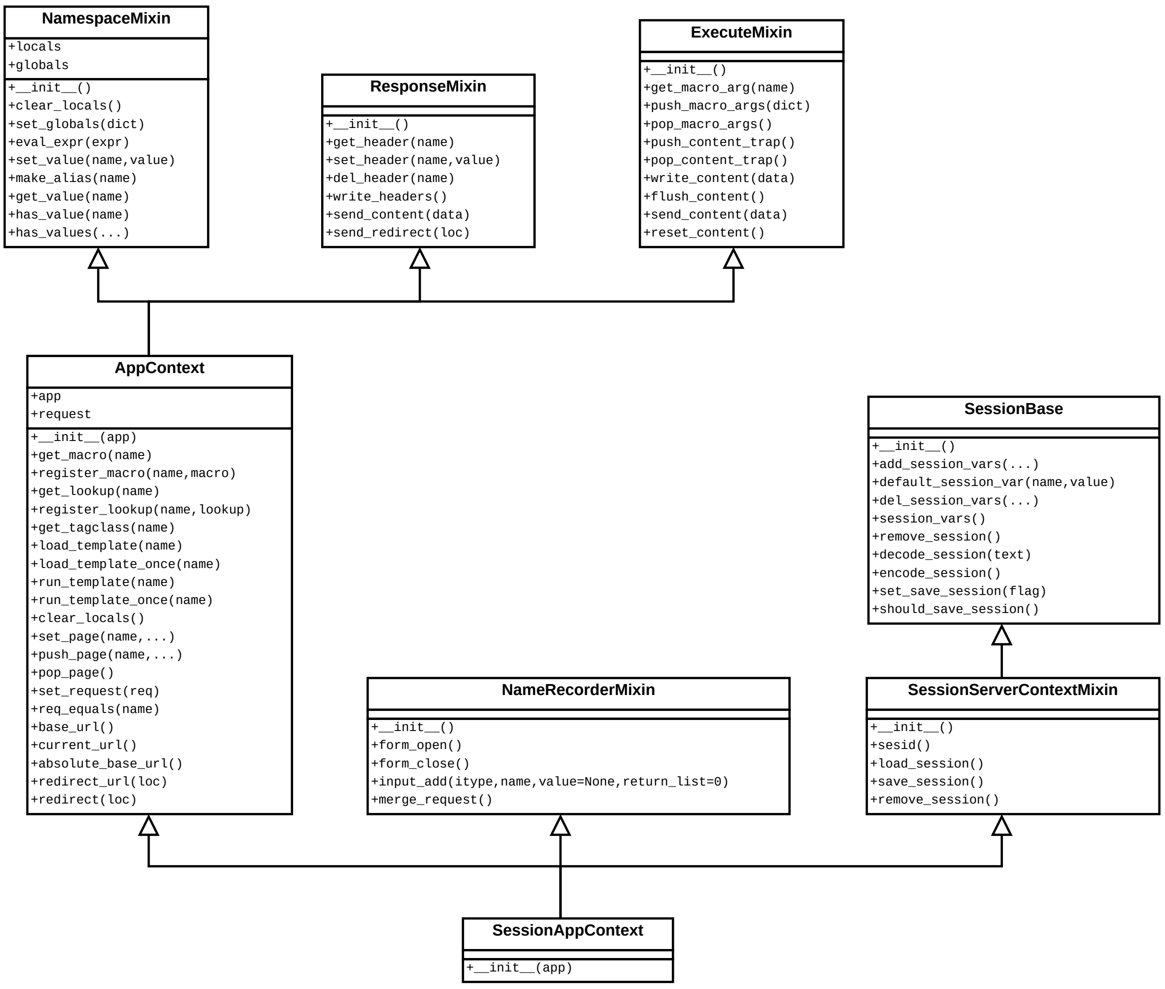


Figure 11.5: The `SessionAppContext` class

11.3.3 The `SessionFileAppContext` Class

The `SessionFileAppContext` class is intended to be used for applications which store state at the server. An inheritance diagram illustrates the relationship to the `SimpleAppContext` class described above.

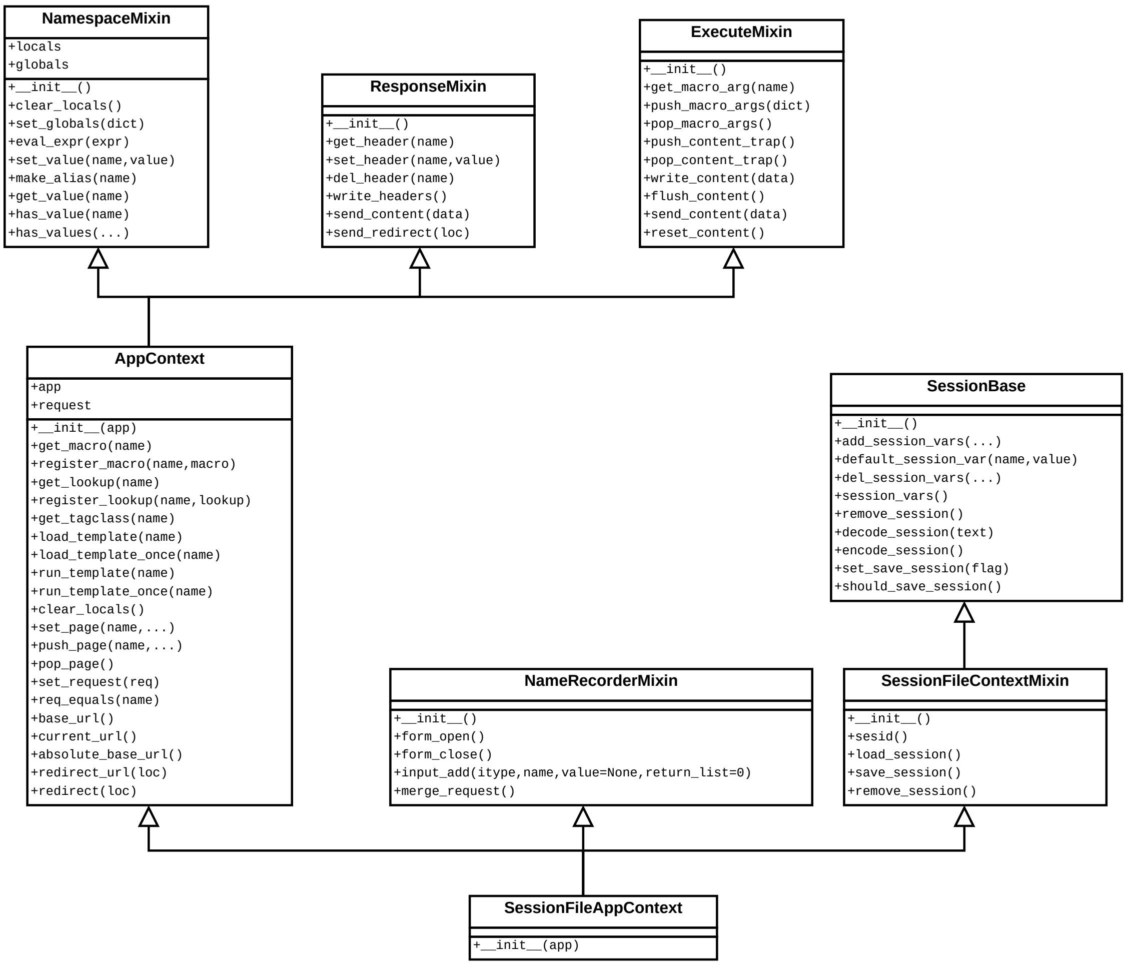


Figure 11.6: The `SessionFileAppContext` class

The methods available in `SessionFileAppContext` and the location of their definition are show below.

Method	Mixin
`absolute_base_url()`	`AppContext`
`add_header(name, value)()`	`ResponseMixin`
`add_session_vars(*names)()`	`SessionBase`
`base_url()`	`AppContext`
`clear_active_select()`	`ExecuteMixin`
`clear_locals()`	`AppContext`
`current_url()`	`AppContext`
`decode_session(text)()`	`SessionBase`
`default_session_var(name, value)()`	`SessionBase`
`del_header(name)()`	`ResponseMixin`
`del_session_vars(*names)()`	`SessionBase`
`encode_session()`	`SessionBase`
`eval_expr(expr)()`	`NamespaceMixin`
`flush_content()`	`ExecuteMixin`
`flush_html()`	`ExecuteMixin`
`form_close()`	`NameRecorderMixin`

Continued on next page

Table 11.5 – continued from previous page

form_open()	NameRecorderMixin
get_active_select()	ExecuteMixin
get_header(name)()	ResponseMixin
get_lookup(name)()	AppContext
get_macro(name)()	AppContext
get_macro_arg(name)()	ExecuteMixin
get_tagclass(name)()	AppContext
get_value(name)()	NamespaceMixin
has_value(name)()	NamespaceMixin
has_values(*names)()	NamespaceMixin
input_add(itype, name, unused_value, return_list)()	NameRecorderMixin
load_session()	SessionFileContextMixin
load_template(name)()	AppContext
load_template_once(name)()	AppContext
make_alias(name)()	NamespaceMixin
merge_request()	NameRecorderMixin
merge_vars(*vars)()	NamespaceMixin
parsed_request_uri()	AppContext
pop_content_trap()	ExecuteMixin
pop_macro_args()	ExecuteMixin
pop_page(target_page)()	AppContext
push_content_trap()	ExecuteMixin
push_macro_args(args, defaults)()	ExecuteMixin
push_page(name, *args)()	AppContext
redirect(loc)()	AppContext
redirect_url(loc)()	AppContext
register_lookup(name, lookup)()	AppContext
register_macro(name, macro)()	AppContext
remove_session()	SessionFileContextMixin
req_equals(name)()	AppContext
reset_content()	ExecuteMixin
run_template(name)()	AppContext
run_template_once(name)()	AppContext
save_session()	SessionFileContextMixin
send_content(data)()	ResponseMixin
send_redirect(loc)()	ResponseMixin
sesid()	SessionFileContextMixin
session_vars()	SessionBase
set_active_select(select, value)()	ExecuteMixin
set_globals(dict)()	NamespaceMixin
set_header(name, value)()	ResponseMixin
set_page(name, *args)()	AppContext
set_request(req)()	AppContext
set_save_session(flag)()	SessionBase
set_value(name, value)()	NamespaceMixin
should_save_session()	SessionBase
write_content(data)()	ExecuteMixin
write_headers()	ResponseMixin

Externally the execution context is almost identical to that of the `SimpleAppContext` class. Instead of saving session data in hidden HTML fields, session data is loaded and saved to the servers local file system, which is managed by the application.

The class defines a number of extra methods.

`__init__`(*app*)
When you inherit from the `SessionFileAppContext` class you must call this constructor.

The *app* argument is passed to the `AppContext` constructor.

11.3.4 The `BranchingSessionContext` Class

The `BranchingSessionContext` class is intended to be used with the server-side session application classes. It creates a new session for each interaction with the client, and stores the session identifier in a hidden form field. This allows us to detect when the browser state rolls back (via the browser `back` button), and find the appropriate session context, giving an effect like the client-side `SimpleAppContext` without storing the entire context in a hidden form field.

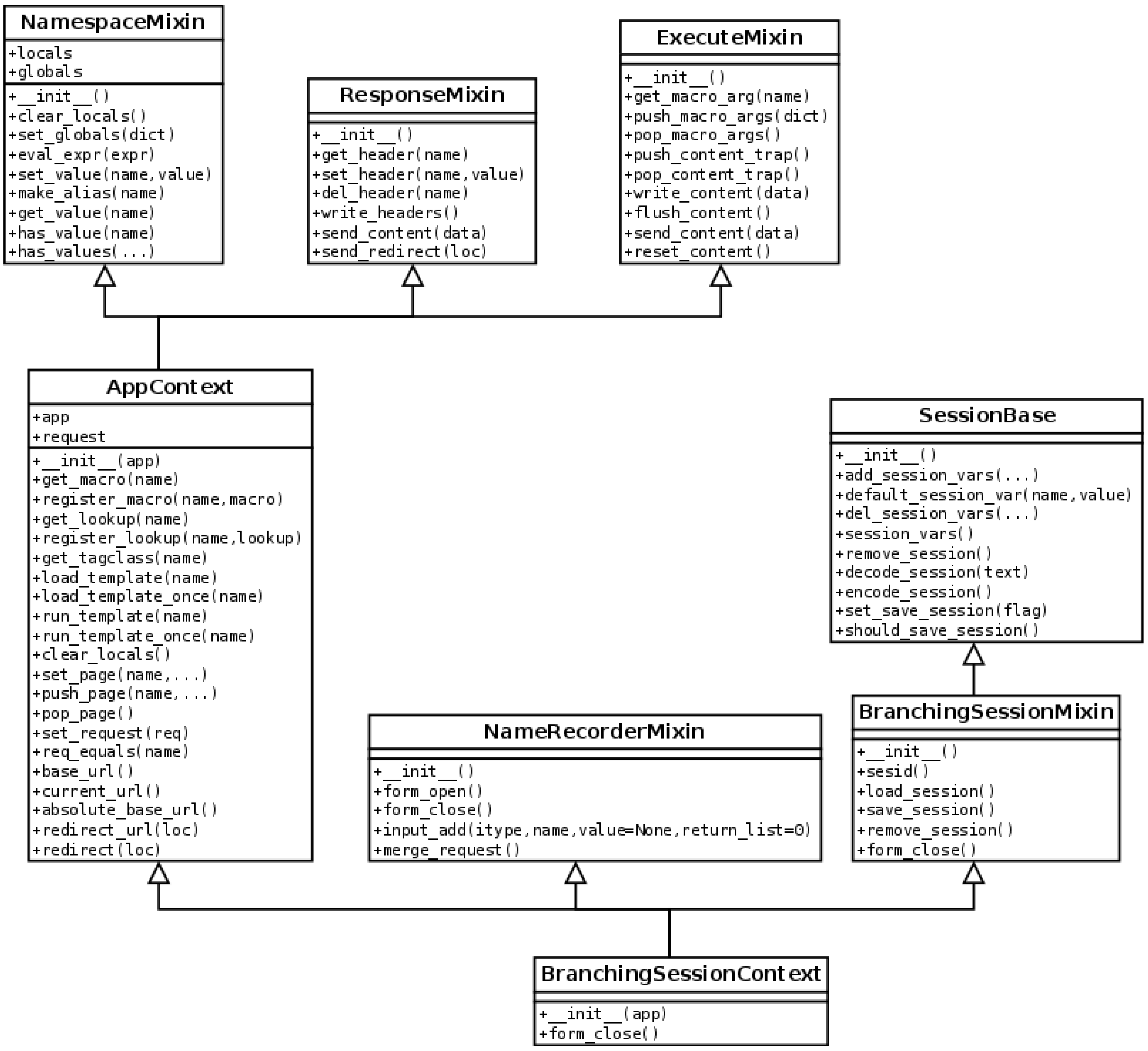


Figure 11.7: The `BranchingSessionContext` class

The methods available in `BranchingSessionContext` and the location of their definition are show below.

Method	Mixin
`absolute_base_url()`	`AppContext`
`add_header(name, value)()`	`ResponseMixin`
`add_session_vars(*names)()`	`SessionBase`
`base_url()`	`AppContext`
`clear_active_select()`	`ExecuteMixin`
`clear_locals()`	`AppContext`

Continued on next page

Table 11.6 – continued from previous page

current_url()	AppContext
decode_session(text)()	SessionBase
default_session_var(name, value)()	SessionBase
del_header(name)()	ResponseMixin
del_session_vars(*names)()	SessionBase
encode_session()	SessionBase
eval_expr(expr)()	NamespaceMixin
flush_content()	ExecuteMixin
flush_html()	ExecuteMixin
form_close()	BranchingSessionContext
form_open()	NameRecorderMixin
get_active_select()	ExecuteMixin
get_header(name)()	ResponseMixin
get_lookup(name)()	AppContext
get_macro(name)()	AppContext
get_macro_arg(name)()	ExecuteMixin
get_tagclass(name)()	AppContext
get_value(name)()	NamespaceMixin
has_value(name)()	NamespaceMixin
has_values(*names)()	NamespaceMixin
input_add(itype, name, unused_value, return_list)()	NameRecorderMixin
load_session()	BranchingSessionMixin
load_template(name)()	AppContext
load_template_once(name)()	AppContext
make_alias(name)()	NamespaceMixin
merge_request()	NameRecorderMixin
merge_vars(*vars)()	NamespaceMixin
parsed_request_uri()	AppContext
pop_content_trap()	ExecuteMixin
pop_macro_args()	ExecuteMixin
pop_page(target_page)()	AppContext
push_content_trap()	ExecuteMixin
push_macro_args(args, defaults)()	ExecuteMixin
push_page(name, *args)()	AppContext
redirect(loc)()	AppContext
redirect_url(loc)()	AppContext
register_lookup(name, lookup)()	AppContext
register_macro(name, macro)()	AppContext
remove_session()	BranchingSessionMixin
req_equals(name)()	AppContext
reset_content()	ExecuteMixin
run_template(name)()	AppContext
run_template_once(name)()	AppContext
save_session()	BranchingSessionMixin
send_content(data)()	ResponseMixin
send_redirect(loc)()	ResponseMixin
sesid()	BranchingSessionMixin
session_vars()	SessionBase
set_active_select(select, value)()	ExecuteMixin
set_globals(dict)()	NamespaceMixin
set_header(name, value)()	ResponseMixin
set_page(name, *args)()	AppContext
set_request(req)()	AppContext
set_save_session(flag)()	SessionBase
set_value(name, value)()	NamespaceMixin
should_save_session()	SessionBase

Continued on next page

Table 11.6 – continued from previous page

`txid()`	`BranchingSessionMixin`
`write_content(data)()`	`ExecuteMixin`
`write_headers()`	`ResponseMixin`

Externally the execution context is almost identical to that of the SimpleAppContext class. Instead of saying the session data in hidden HTML fields, the session identifier is stored in a hidden field, and the session data is saved and loaded from the session server.

The class defines a number of extra methods:

`__init__`(*app*)

When you inherit from the `BranchingSessionContext` class you must call this constructor.

The *app* argument is passed to the `AppContext` constructor.

`form_close`()

Invokes the `form_close()` method of the `NameRecorderMixin` class and `encode_session()` of the `BranchingSessionMixin` class.

11.4 The `Application` Base Class

The `Application` class is the base class for all Albatross application objects.

The class inherits from the `ResourceMixin` class to allow all application resources to be loaded once and used for every browser request. The `AppContext` class directs all resource related execution context method here.

The methods available in `Application` and the location of their definition are show below.

Method	Mixin
`base_url()`	`Application`
`discard_file_resources(filename)()`	`ResourceMixin`
`format_exception()`	`Application`
`get_lookup(name)()`	`ResourceMixin`
`get_macro(name)()`	`ResourceMixin`
`get_tagclass(name)()`	`ResourceMixin`
`handle_exception(ctx, req)()`	`Application`
`load_session(ctx)()`	`Application`
`merge_request(ctx)()`	`Application`
`pickle_sign(text)()`	`Application`
`pickle_unsign(text)()`	`Application`
`register_lookup(name, lookup)()`	`ResourceMixin`
`register_macro(name, macro)()`	`ResourceMixin`
`register_tagclasses(*tags)()`	`ResourceMixin`
`remove_session(ctx)()`	`Application`
`run(req)()`	`Application`
`save_session(ctx)()`	`Application`
`template_traceback(tb)()`	`Application`
`validate_request(ctx)()`	`Application`

The `Application` class introduces a number of new methods.

`__init__`(*base_url*)

When you inherit from the `Application` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags.

`base_url`()

Returns the *base_url* argument which was passed to the constructor.

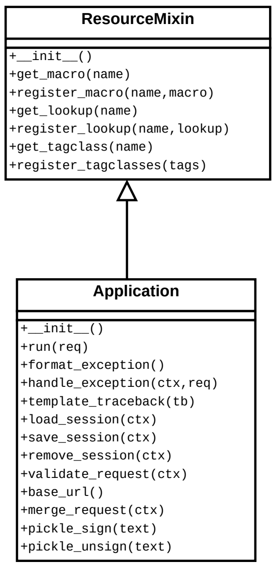


Figure 11.8: The `Application` class

run (*req*)
: Implements the standard application run sequence as described in the *Albatross Application Model* section. The browser request passed as the *req* argument is attached to the execution context as soon as the context has been created.

  If an exception is caught then the `handle_exception()` method is called passing the *req* argument.

format_exception ()
: Retrieves the current exception from `sys.exc_info()` then formats and returns the standard Python traceback and a template interpreter traceback.

handle_exception (*ctx, req*)
: This implements the default exception handling for applications. The *req* argument is the browser request which was passed to the `run()` method.

  The method calls the `format_exception()` method to construct a standard Python traceback and a template traceback. A temporary execution context is then created, the Python traceback is saved in the `locals.python_exc` value, and the template traceback in the `locals.html_exc` value.

  The method then tries to load the `'traceback.html'` template file and execute it with the temporary execution context. This gives you the ability to control the presentation and reporting of exceptions.

  If any exceptions are raised during the execution of `'traceback.html'` the method writes both formatted exceptions as a `<pre>` formatted browser response.

template_traceback (*tb*)
: Generates a template interpreter traceback from the Python stack trace in the *tb* argument.

load_session (*ctx*)
: Calls the `load_session()` method of the execution context in the *ctx* argument.

save_session (*ctx*)
: Calls the `save_session()` method of the execution context in the *ctx* argument.

remove_session (*ctx*)
: Calls the `remove_session()` method of the execution context in the *ctx* argument.

validate_request (*ctx*)
: Returns `TRUE`.

  You should override this method in your application object if you need to validate browser requests before processing them. Returning False will prevent the browser request being merged into the local namespace and the page_process logic will not be called (see also the description of the Albatross Application Model in *Albatross Application Model*).

pickle_sign (*text*)
Returns an empty string to prevent insecure pickles being sent to the browser. This is overridden in the `PickleSignMixin` class.

pickle_unsign (*text*)
Returns an empty string to prevent insecure pickles being accepted from the browser. This is overridden in the `PickleSignMixin` class.

merge_request (*ctx*)
Calls the `merge_request()` method of the execution context.

11.5 Application Classes:

11.5.1 The SimpleApp Class

The `SimpleApp` class is intended for use in monolithic applications (page objects instead of page modules). An inheritance diagram illustrates the relationship to the `SimpleAppContext` class described above.

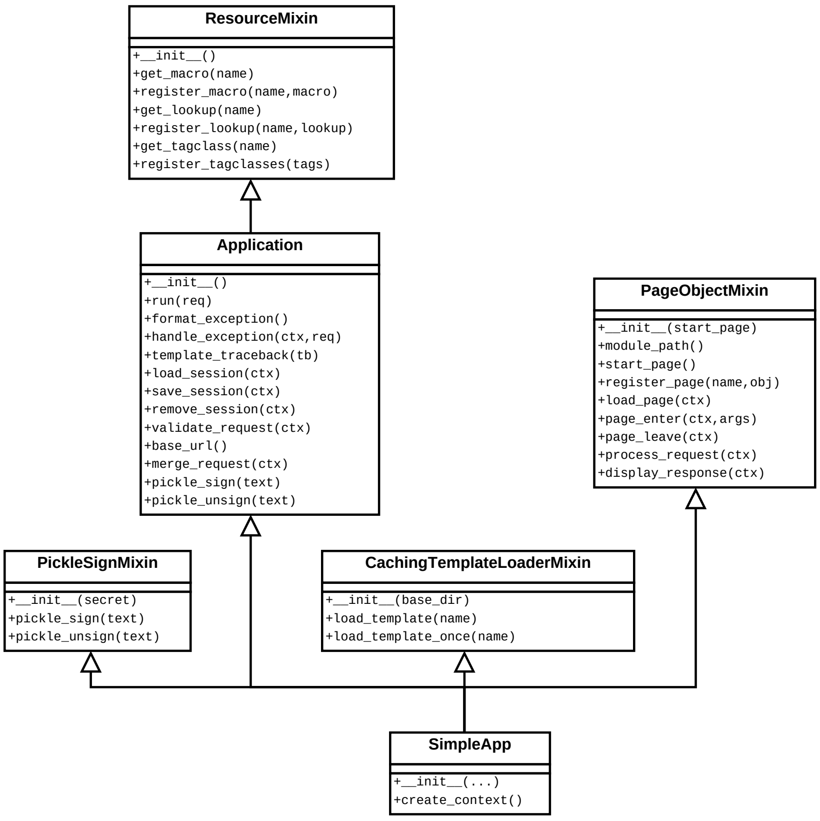


Figure 11.9: The `SimpleApp` class

The methods available in `SimpleApp` and the location of their definition are show below.

Method	Mixin
`base_url()`	`Application`
`create_context()`	`SimpleApp`
`discard_file_resources(filename)()`	`ResourceMixin`
`display_response(ctx)()`	`PageObjectMixin`

Continued on next page

Table 11.7 – continued from previous page

format_exception()	Application
get_lookup(name)()	ResourceMixin
get_macro(name)()	ResourceMixin
get_tagclass(name)()	ResourceMixin
handle_exception(ctx, req)()	Application
is_page_module(name)()	PageObjectMixin
load_page(ctx)()	PageObjectMixin
load_session(ctx)()	Application
load_template(name)()	CachingTemplateLoaderMixin
load_template_once(name)()	CachingTemplateLoaderMixin
merge_request(ctx)()	Application
page_enter(ctx, args)()	PageObjectMixin
page_leave(ctx)()	PageObjectMixin
pickle_sign(text)()	PickleSignMixin
pickle_unsign(text)()	PickleSignMixin
process_request(ctx)()	PageObjectMixin
register_lookup(name, lookup)()	ResourceMixin
register_macro(name, macro)()	ResourceMixin
register_page(name, obj)()	PageObjectMixin
register_tagclasses(*tags)()	ResourceMixin
remove_session(ctx)()	Application
run(req)()	Application
save_session(ctx)()	Application
start_page()	PageObjectMixin
template_traceback(tb)()	Application
validate_request(ctx)()	Application

The `SimpleApp` class defines the following methods:

`__init__`(*base_url, template_path, start_page, secret*)
When you inherit from the `SimpleApp` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags. The *template_path* defines the root directory where template files are loaded from. The *start_page* identifies the first page that will be served up in a new browser session. The *secret* argument is used to sign all pickles sent to the browser.

`create_context`()
Returns a new instance of the `SimpleAppContext` class.

11.5.2 The `SimpleSessionApp` Class

The `SimpleSessionApp` class is intended for use in monolithic applications (page objects instead of page modules). Session state is stored at the server.

The methods available in `SimpleSessionApp` and the location of their definition are show below.

Method	Mixin
base_url()	Application
create_context()	SimpleSessionApp
del_session(sesid)()	SessionServerAppMixin
discard_file_resources(filename)()	ResourceMixin
display_response(ctx)()	PageObjectMixin
format_exception()	Application
get_lookup(name)()	ResourceMixin
get_macro(name)()	ResourceMixin
get_session(sesid)()	SessionServerAppMixin

Continued on next page

Table 11.8 – continued from previous page

get_tagclass(name)()	ResourceMixin
handle_exception(ctx, req)()	Application
is_page_module(name)()	PageObjectMixin
load_page(ctx)()	PageObjectMixin
load_session(ctx)()	Application
load_template(name)()	CachingTemplateLoaderMixin
load_template_once(name)()	CachingTemplateLoaderMixin
merge_request(ctx)()	Application
new_session()	SessionServerAppMixin
page_enter(ctx, args)()	PageObjectMixin
page_leave(ctx)()	PageObjectMixin
pickle_sign(text)()	PickleSignMixin
pickle_unsign(text)()	PickleSignMixin
process_request(ctx)()	PageObjectMixin
put_session(sesid, text)()	SessionServerAppMixin
register_lookup(name, lookup)()	ResourceMixin
register_macro(name, macro)()	ResourceMixin
register_page(name, obj)()	PageObjectMixin
register_tagclasses(*tags)()	ResourceMixin
remove_session(ctx)()	Application
run(req)()	Application
save_session(ctx)()	Application
ses_age()	SessionServerAppMixin
ses_appid()	SessionServerAppMixin
start_page()	PageObjectMixin
template_traceback(tb)()	Application
validate_request(ctx)()	Application

The SimpleSessionApp class defines the following methods:

__init__(*base_url, template_path, start_page, secret, session_appid, [session_server ``= 'localhost'``], [server_port ``= 34343``], [session_age ``= 1800``]*)

When you inherit from the SimpleSessionApp class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the <al-a> and <al-form> tags. The *template_path* defines the root directory where template files are loaded from. The *start_page* identifies the first page that will be served up in a new browser session. The *secret* argument is used to sign all pickles sent to the browser.

The *session_appid* argument identifies the session application at the session server. Multiple applications can share sessions by using the same identifier here. The *session_server* argument defines the host where the session server is running, it defaults to localhost. The *server_port* defines the session server port, it defaults to 34343. The *session_age* argument defines the number of seconds that an idle session will be kept, it defaults to 1800.

create_context()

Returns a new instance of the SessionAppContext class.

11.5.3 The SimpleSessionFileApp Class

The SimpleSessionFileApp class is intended for use in monolithic applications (page objects instead of page modules). Session state is stored in the file system at the server.

The methods available in SimpleSessionFileApp and the location of their definition are show below.

Method	Mixin

Continued on next page

Table 11.9 – continued from previous page

base_url()	Application
create_context()	SimpleSessionFileApp
del_session(sesid)()	SessionFileAppMixin
discard_file_resources(filename)()	ResourceMixin
display_response(ctx)()	PageObjectMixin
format_exception()	Application
get_lookup(name)()	ResourceMixin
get_macro(name)()	ResourceMixin
get_session(sesid)()	SessionFileAppMixin
get_tagclass(name)()	ResourceMixin
handle_exception(ctx, req)()	Application
is_page_module(name)()	PageObjectMixin
load_page(ctx)()	PageObjectMixin
load_session(ctx)()	Application
load_template(name)()	CachingTemplateLoaderMixin
load_template_once(name)()	CachingTemplateLoaderMixin
merge_request(ctx)()	Application
new_session()	SessionFileAppMixin
page_enter(ctx, args)()	PageObjectMixin
page_leave(ctx)()	PageObjectMixin
pickle_sign(text)()	PickleSignMixin
pickle_unsign(text)()	PickleSignMixin
process_request(ctx)()	PageObjectMixin
put_session(sesid, text)()	SessionFileAppMixin
register_lookup(name, lookup)()	ResourceMixin
register_macro(name, macro)()	ResourceMixin
register_page(name, obj)()	PageObjectMixin
register_tagclasses(*tags)()	ResourceMixin
remove_session(ctx)()	Application
run(req)()	Application
save_session(ctx)()	Application
ses_age()	SessionFileAppMixin
ses_appid()	SessionFileAppMixin
start_page()	PageObjectMixin
template_traceback(tb)()	Application
validate_request(ctx)()	Application

The `SimpleSessionFileApp` class defines the following methods:

`__init__`(*base_url, template_path, start_page, secret, session_appid, session_dir*)
When you inherit from the `SimpleSessionFileApp` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags. The *template_path* defines the root directory where template files are loaded from. The *start_page* identifies the first page that will be served up in a new browser session. The *secret* argument is used to sign all pickles sent to the browser.

The *session_appid* argument identifies the session application in the browser cookie. Multiple applications can share sessions by using the same identifier here. The *session_dir* argument defines the directory in which the application will store session files.

`create_context`()
Returns a new instance of the `SessionFileAppContext` class.

11.5.4 The `ModularApp` Class

The `ModularApp` class is intended for use in applications which define page code in a collection of Python modules.

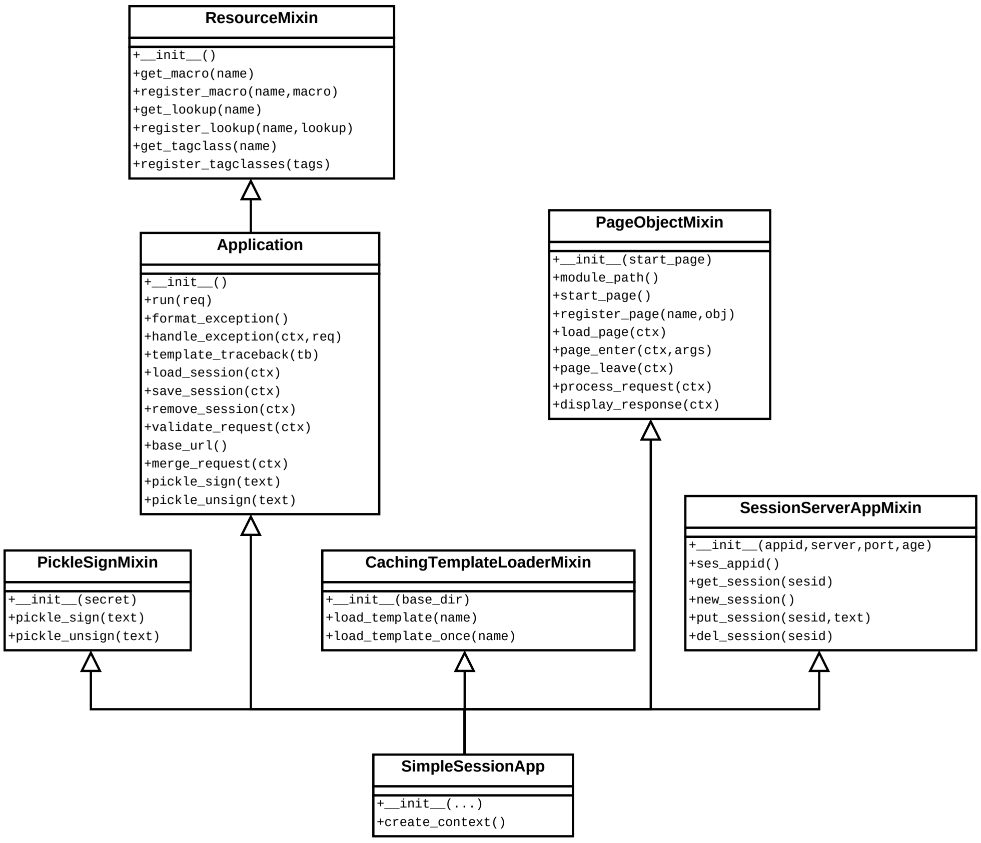


Figure 11.10: The `SimpleSessionApp` class

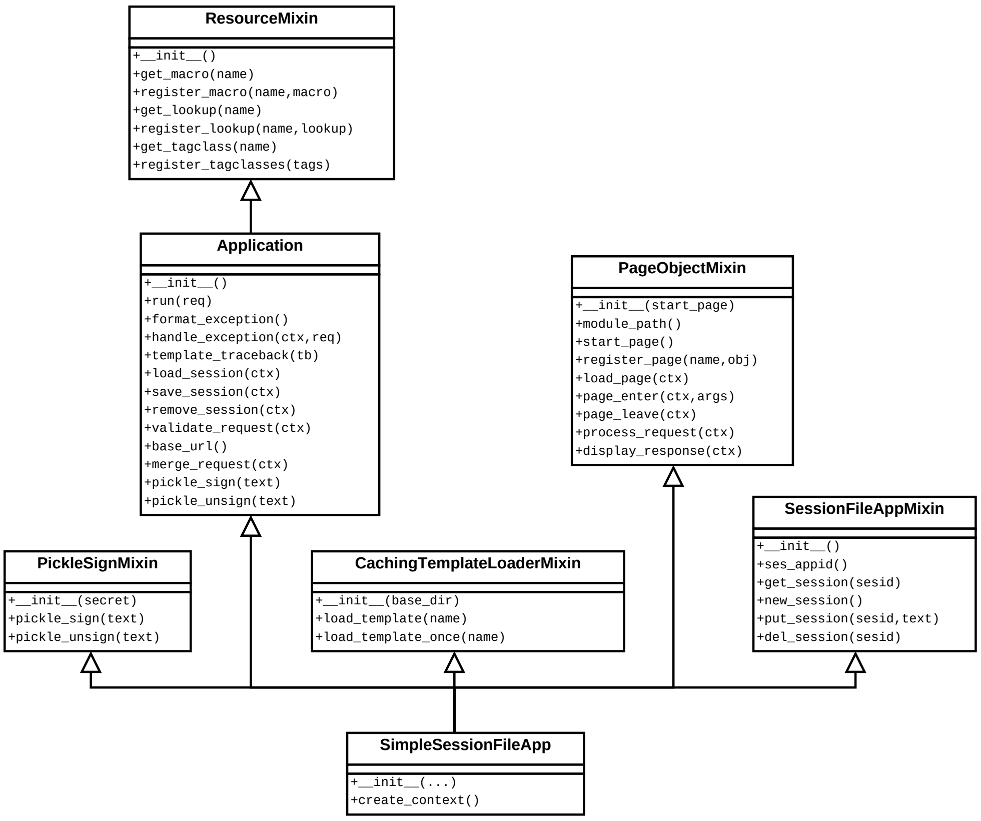


Figure 11.11: The `SimpleSessionFileApp` class

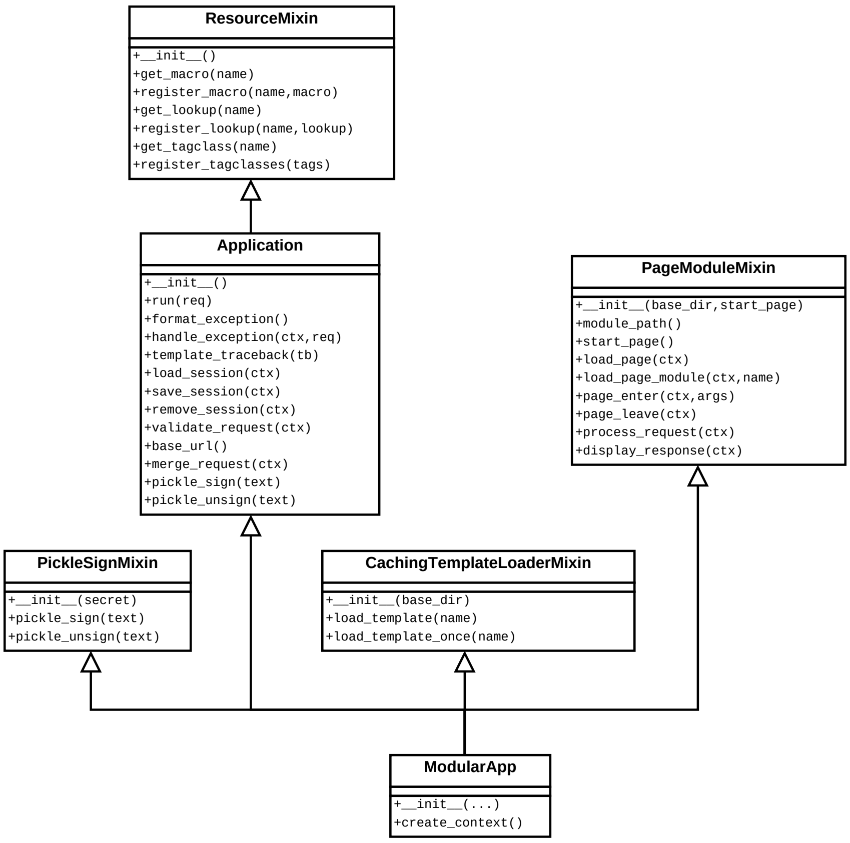


Figure 11.12: The `ModularApp` class

The methods available in `ModularApp` and the location of their definition are show below.

Method	Mixin
`base_url()`	`Application`
`create_context()`	`ModularApp`
`discard_file_resources(filename)()`	`ResourceMixin`
`display_response(ctx)()`	`PageModuleMixin`
`format_exception()`	`Application`
`get_lookup(name)()`	`ResourceMixin`
`get_macro(name)()`	`ResourceMixin`
`get_tagclass(name)()`	`ResourceMixin`
`handle_exception(ctx, req)()`	`Application`
`is_page_module(name)()`	`PageModuleMixin`
`load_page(ctx)()`	`PageModuleMixin`
`load_page_module(ctx, name)()`	`PageModuleMixin`
`load_session(ctx)()`	`Application`
`load_template(name)()`	`CachingTemplateLoaderMixin`
`load_template_once(name)()`	`CachingTemplateLoaderMixin`
`merge_request(ctx)()`	`Application`
`module_path()`	`PageModuleMixin`
`page_enter(ctx, args)()`	`PageModuleMixin`
`page_leave(ctx)()`	`PageModuleMixin`
`pickle_sign(text)()`	`PickleSignMixin`
`pickle_unsign(text)()`	`PickleSignMixin`
`process_request(ctx)()`	`PageModuleMixin`
`register_lookup(name, lookup)()`	`ResourceMixin`
`register_macro(name, macro)()`	`ResourceMixin`
`register_tagclasses(*tags)()`	`ResourceMixin`

Continued on next page

Table 11.10 – continued from previous page

`remove_session(ctx)()`	`Application`
`run(req)()`	`Application`
`save_session(ctx)()`	`Application`
`start_page()`	`PageModuleMixin`
`template_traceback(tb)()`	`Application`
`validate_request(ctx)()`	`Application`

The `ModularApp` class defines the following methods:

`__init__`(*base_url, module_path, template_path, start_page, secret*)
When you inherit from the `ModularApp` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags. The *module_path* argument defines the root directory where page modules are loaded from. The *template_path* argument defines the root directory where template files are loaded from. The *start_page* identifies the first page that will be served up in a new browser session. The *secret* argument is used to sign all pickles sent to the browser.

`create_context`()
Returns a new instance of the `SimpleAppContext` class.

11.5.5 The `ModularSessionApp` Class

The `ModularSessionApp` class is intended for use in applications which define page code in a collection of Python modules. Session state is stored at the server.

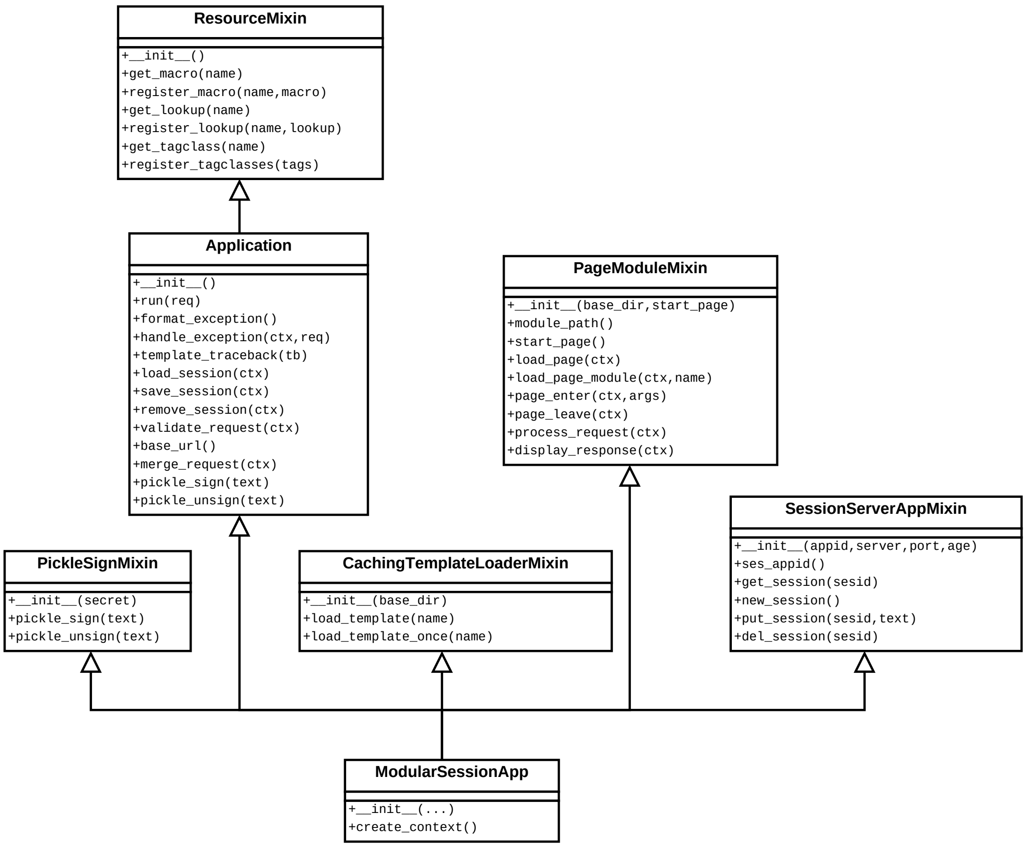


Figure 11.13: The `ModularSessionApp` class

The methods available in `ModularSessionApp` and the location of their definition are show below.

Method	Mixin
`base_url()`	`Application`
`create_context()`	`ModularSessionApp`
`del_session(sesid)()`	`SessionServerAppMixin`
`discard_file_resources(filename)()`	`ResourceMixin`
`display_response(ctx)()`	`PageModuleMixin`
`format_exception()`	`Application`
`get_lookup(name)()`	`ResourceMixin`
`get_macro(name)()`	`ResourceMixin`
`get_session(sesid)()`	`SessionServerAppMixin`
`get_tagclass(name)()`	`ResourceMixin`
`handle_exception(ctx, req)()`	`Application`
`is_page_module(name)()`	`PageModuleMixin`
`load_page(ctx)()`	`PageModuleMixin`
`load_page_module(ctx, name)()`	`PageModuleMixin`
`load_session(ctx)()`	`Application`
`load_template(name)()`	`CachingTemplateLoaderMixin`
`load_template_once(name)()`	`CachingTemplateLoaderMixin`
`merge_request(ctx)()`	`Application`
`module_path()`	`PageModuleMixin`
`new_session()`	`SessionServerAppMixin`
`page_enter(ctx, args)()`	`PageModuleMixin`
`page_leave(ctx)()`	`PageModuleMixin`
`pickle_sign(text)()`	`PickleSignMixin`
`pickle_unsign(text)()`	`PickleSignMixin`
`process_request(ctx)()`	`PageModuleMixin`
`put_session(sesid, text)()`	`SessionServerAppMixin`
`register_lookup(name, lookup)()`	`ResourceMixin`
`register_macro(name, macro)()`	`ResourceMixin`
`register_tagclasses(*tags)()`	`ResourceMixin`
`remove_session(ctx)()`	`Application`
`run(req)()`	`Application`
`save_session(ctx)()`	`Application`
`ses_age()`	`SessionServerAppMixin`
`ses_appid()`	`SessionServerAppMixin`
`start_page()`	`PageModuleMixin`
`template_traceback(tb)()`	`Application`
`validate_request(ctx)()`	`Application`

The `ModularSessionApp` class defines the following methods:

`__init__`(*base_url, module_path, template_path, start_page, secret, session_appid, [session_server ''= 'localhost'''], [server_port ''= 34343''], [session_age ''= 1800'']*)

When you inherit from the `ModularSessionApp` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags. The *module_path* argument defines the root directory where page modules are loaded from. The *template_path* argument defines the root directory where template files are loaded from. The *start_page* identifies the first page that will be served up in a new browser session. The *secret* argument is used to sign all pickles sent to the browser.

The *session_appid* argument identifies the session application at the session server. Multiple applications can share sessions by using the same identifier here. The *session_server* argument defines the host where the session server is running, it defaults to `localhost`. The *server_port* defines the session server port, it defaults to `34343`. The *session_age* argument defines the number of seconds that an idle session will be kept, it defaults to `1800`.

`create_context`()

Returns a new instance of the `SessionAppContext` class.

11.5.6 The `ModularSessionFileApp` Class

The `ModularSessionFileApp` class is intended for use in applications which define page code in a collection of Python modules. Session state is stored in the file system at the server.

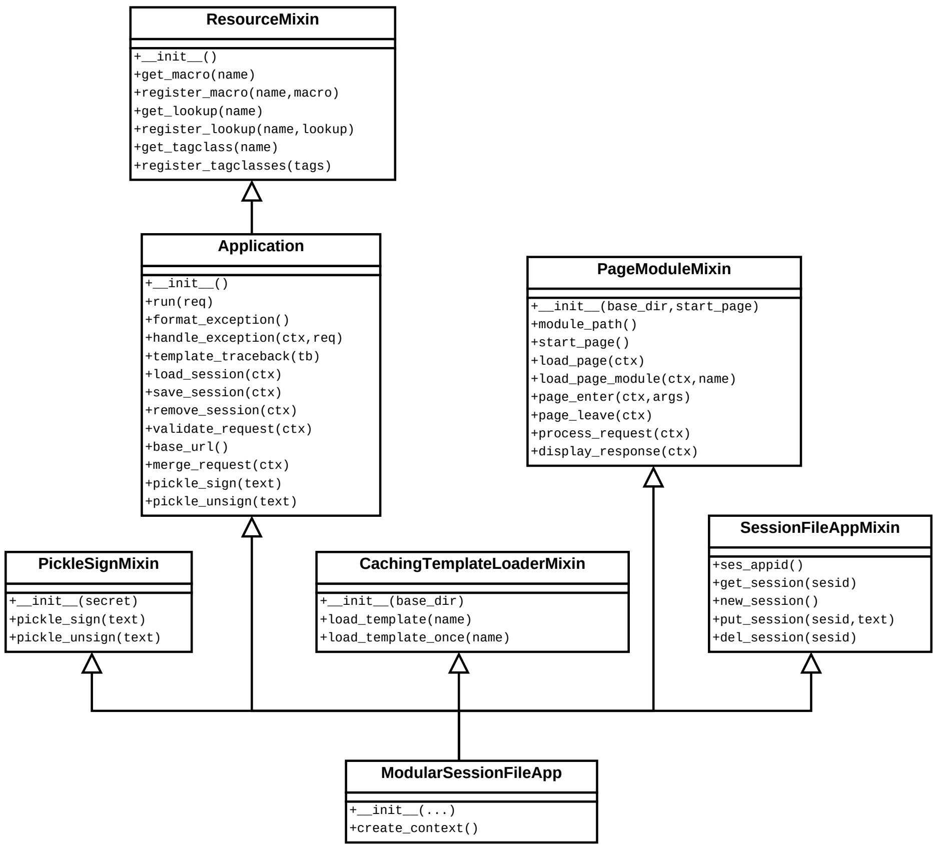


Figure 11.14: The `ModularSessionFileApp` class

The methods available in `ModularSessionFileApp` and the location of their definition are show below.

Method	Mixin
`base_url()`	`Application`
`create_context()`	`ModularSessionFileApp`
`del_session(sesid)()`	`SessionFileAppMixin`
`discard_file_resources(filename)()`	`ResourceMixin`
`display_response(ctx)()`	`PageModuleMixin`
`format_exception()`	`Application`
`get_lookup(name)()`	`ResourceMixin`
`get_macro(name)()`	`ResourceMixin`
`get_session(sesid)()`	`SessionFileAppMixin`
`get_tagclass(name)()`	`ResourceMixin`
`handle_exception(ctx, req)()`	`Application`
`is_page_module(name)()`	`PageModuleMixin`
`load_page(ctx)()`	`PageModuleMixin`
`load_page_module(ctx, name)()`	`PageModuleMixin`
`load_session(ctx)()`	`Application`
`load_template(name)()`	`CachingTemplateLoaderMixin`
`load_template_once(name)()`	`CachingTemplateLoaderMixin`
`merge_request(ctx)()`	`Application`
`module_path()`	`PageModuleMixin`
`new_session()`	`SessionFileAppMixin`

Continued on next page

Table 11.12 – continued from previous page

`page_enter(ctx, args)()`	`PageModuleMixin`
`page_leave(ctx)()`	`PageModuleMixin`
`pickle_sign(text)()`	`PickleSignMixin`
`pickle_unsign(text)()`	`PickleSignMixin`
`process_request(ctx)()`	`PageModuleMixin`
`put_session(sesid, text)()`	`SessionFileAppMixin`
`register_lookup(name, lookup)()`	`ResourceMixin`
`register_macro(name, macro)()`	`ResourceMixin`
`register_tagclasses(*tags)()`	`ResourceMixin`
`remove_session(ctx)()`	`Application`
`run(req)()`	`Application`
`save_session(ctx)()`	`Application`
`ses_age()`	`SessionFileAppMixin`
`ses_appid()`	`SessionFileAppMixin`
`start_page()`	`PageModuleMixin`
`template_traceback(tb)()`	`Application`
`validate_request(ctx)()`	`Application`

The `ModularSessionFileApp` class defines the following methods:

`__init__`(*base_url, module_path, template_path, start_page, secret, session_appid, session_dir*)
When you inherit from the `ModularSessionFileApp` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags. The *module_path* argument defines the root directory where page modules are loaded from. The *template_path* argument defines the root directory where template files are loaded from. The *start_page* identifies the first page that will be served up in a new browser session. The *secret* argument is used to sign all pickles sent to the browser.

The *session_appid* argument identifies the session application in the browser cookie. Multiple applications can share sessions by using the same identifier here. The *session_dir* argument defines the directory in which the application will store session files.

`create_context`()
Returns a new instance of the `SessionFileAppContext` class.

11.5.7 The `RandomModularApp` Class

The `RandomModularApp` class is intended for use in applications which define page code in a collection of Python modules which are randomly accessed via the URI in the browser request.

The methods available in `RandomModularApp` and the location of their definition are show below.

Method	Mixin
`base_url()`	`Application`
`create_context()`	`RandomModularApp`
`discard_file_resources(filename)()`	`ResourceMixin`
`display_response(ctx)()`	`RandomPageModuleMixin`
`format_exception()`	`Application`
`get_lookup(name)()`	`ResourceMixin`
`get_macro(name)()`	`ResourceMixin`
`get_page_from_uri(ctx, uri)()`	`RandomPageModuleMixin`
`get_tagclass(name)()`	`ResourceMixin`
`handle_exception(ctx, req)()`	`Application`
`is_page_module(name)()`	`PageModuleMixin`
`load_badurl_template(ctx)()`	`RandomPageModuleMixin`
`load_page(ctx)()`	`RandomPageModuleMixin`

Continued on next page

Table 11.13 – continued from previous page

load_page_module(ctx, name)()	PageModuleMixin
load_session(ctx)()	Application
load_template(name)()	CachingTemplateLoaderMixin
load_template_once(name)()	CachingTemplateLoaderMixin
merge_request(ctx)()	Application
module_path()	PageModuleMixin
page_enter(ctx)()	RandomPageModuleMixin
page_leave(ctx)()	PageModuleMixin
pickle_sign(text)()	PickleSignMixin
pickle_unsign(text)()	PickleSignMixin
process_request(ctx)()	RandomPageModuleMixin
register_lookup(name, lookup)()	ResourceMixin
register_macro(name, macro)()	ResourceMixin
register_tagclasses(*tags)()	ResourceMixin
remove_session(ctx)()	Application
run(req)()	Application
save_session(ctx)()	Application
start_page()	PageModuleMixin
template_traceback(tb)()	Application
validate_request(ctx)()	Application

The `RandomModularApp` class defines the following methods:

`__init__`(*base_url, page_path, start_page, secret*)

When you inherit from the `RandomModularApp` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags. The *page_path* argument defines the root directory where page modules and template files are loaded from. The *start_page* identifies the page that will be served up when a page identifier cannot be determined from the URI in the browser request. The *secret* argument is used to sign all pickles sent to the browser.

`create_context`()

Returns a new instance of the `SimpleAppContext` class.

11.5.8 The `RandomModularSessionApp` Class

The `RandomModularSessionApp` class is intended for use in applications which define page code in a collection of Python modules which are randomly accessed via the URI in the browser request. Session state is stored at the server.

The methods available in `RandomModularSessionApp` and the location of their definition are show below.

Method	Mixin
base_url()	Application
create_context()	RandomModularSessionApp
del_session(sesid)()	SessionServerAppMixin
discard_file_resources(filename)()	ResourceMixin
display_response(ctx)()	RandomPageModuleMixin
format_exception()	Application
get_lookup(name)()	ResourceMixin
get_macro(name)()	ResourceMixin
get_page_from_uri(ctx, uri)()	RandomPageModuleMixin
get_session(sesid)()	SessionServerAppMixin
get_tagclass(name)()	ResourceMixin
handle_exception(ctx, req)()	Application
is_page_module(name)()	PageModuleMixin
load_badurl_template(ctx)()	RandomPageModuleMixin

Continued on next page

Table 11.14 – continued from previous page

`load_page(ctx)()`	`RandomPageModuleMixin`
`load_page_module(ctx, name)()`	`PageModuleMixin`
`load_session(ctx)()`	`Application`
`load_template(name)()`	`CachingTemplateLoaderMixin`
`load_template_once(name)()`	`CachingTemplateLoaderMixin`
`merge_request(ctx)()`	`Application`
`module_path()`	`PageModuleMixin`
`new_session()`	`SessionServerAppMixin`
`page_enter(ctx)()`	`RandomPageModuleMixin`
`page_leave(ctx)()`	`PageModuleMixin`
`pickle_sign(text)()`	`PickleSignMixin`
`pickle_unsign(text)()`	`PickleSignMixin`
`process_request(ctx)()`	`RandomPageModuleMixin`
`put_session(sesid, text)()`	`SessionServerAppMixin`
`register_lookup(name, lookup)()`	`ResourceMixin`
`register_macro(name, macro)()`	`ResourceMixin`
`register_tagclasses(*tags)()`	`ResourceMixin`
`remove_session(ctx)()`	`Application`
`run(req)()`	`Application`
`save_session(ctx)()`	`Application`
`ses_age()`	`SessionServerAppMixin`
`ses_appid()`	`SessionServerAppMixin`
`start_page()`	`PageModuleMixin`
`template_traceback(tb)()`	`Application`
`validate_request(ctx)()`	`Application`

The `RandomModularSessionApp` class defines the following methods:

`__init__`(*base_url, page_path, start_page, secret, session_appid, [session_server ``= 'localhost'``], [server_port ``= 34343``], [session_age ``= 1800``]*)

When you inherit from the `RandomModularSessionApp` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags. The *page_path* argument defines the root directory where page modules and template files are loaded from. The *start_page* identifies the page that will be served up when a page identifier cannot be determined from the URI in the browser request. The *secret* argument is used to sign all pickles sent to the browser.

The *session_appid* argument identifies the session application at the session server. Multiple applications can share sessions by using the same identifier here. The *session_server* argument defines the host where the session server is running, it defaults to `localhost`. The *server_port* defines the session server port, it defaults to `34343`. The *session_age* argument defines the number of seconds that an idle session will be kept, it defaults to `1800`.

`create_context`()

Returns a new instance of the `SessionAppContext` class.

11.5.9 The `RandomModularSessionFileApp` Class

The `RandomModularSessionFileApp` class is intended for use in applications which define page code in a collection of Python modules which are randomly accessed via the URI in the browser request. Session state is stored in the file system at the server.

The methods available in `RandomModularSessionFileApp` and the location of their definition are show below.

Method	Mixin

Continued on next page

Table 11.15 – continued from previous page

`base_url()`	`Application`
`create_context()`	`RandomModularSessionFileApp`
`del_session(sesid)()`	`SessionFileAppMixin`
`discard_file_resources(filename)()`	`ResourceMixin`
`display_response(ctx)()`	`RandomPageModuleMixin`
`format_exception()`	`Application`
`get_lookup(name)()`	`ResourceMixin`
`get_macro(name)()`	`ResourceMixin`
`get_page_from_uri(ctx, uri)()`	`RandomPageModuleMixin`
`get_session(sesid)()`	`SessionFileAppMixin`
`get_tagclass(name)()`	`ResourceMixin`
`handle_exception(ctx, req)()`	`Application`
`is_page_module(name)()`	`PageModuleMixin`
`load_badurl_template(ctx)()`	`RandomPageModuleMixin`
`load_page(ctx)()`	`RandomPageModuleMixin`
`load_page_module(ctx, name)()`	`PageModuleMixin`
`load_session(ctx)()`	`Application`
`load_template(name)()`	`CachingTemplateLoaderMixin`
`load_template_once(name)()`	`CachingTemplateLoaderMixin`
`merge_request(ctx)()`	`Application`
`module_path()`	`PageModuleMixin`
`new_session()`	`SessionFileAppMixin`
`page_enter(ctx)()`	`RandomPageModuleMixin`
`page_leave(ctx)()`	`PageModuleMixin`
`pickle_sign(text)()`	`PickleSignMixin`
`pickle_unsign(text)()`	`PickleSignMixin`
`process_request(ctx)()`	`RandomPageModuleMixin`
`put_session(sesid, text)()`	`SessionFileAppMixin`
`register_lookup(name, lookup)()`	`ResourceMixin`
`register_macro(name, macro)()`	`ResourceMixin`
`register_tagclasses(*tags)()`	`ResourceMixin`
`remove_session(ctx)()`	`Application`
`run(req)()`	`Application`
`save_session(ctx)()`	`Application`
`ses_age()`	`SessionFileAppMixin`
`ses_appid()`	`SessionFileAppMixin`
`start_page()`	`PageModuleMixin`
`template_traceback(tb)()`	`Application`
`validate_request(ctx)()`	`Application`

The `RandomModularSessionFileApp` class defines the following methods:

`__init__`(*base_url, page_path, start_page, secret, session_appid, session_dir*)
When you inherit from the `RandomModularSessionFileApp` class you must call this constructor.

The *base_url* argument is used as the base for URLs produced by the `<al-a>` and `<al-form>` tags. The *page_path* argument defines the root directory where page modules and template files are loaded from. The *start_page* identifies the page that will be served up when a page identifier cannot be determined from the URI in the browser request. The *secret* argument is used to sign all pickles sent to the browser.

The *session_appid* argument identifies the session application at the session server. Multiple applications can share sessions by using the same identifier here. The *session_dir* argument defines the directory in which the application will store session files.

`create_context`()
Returns a new instance of the `SessionFileAppContext` class.

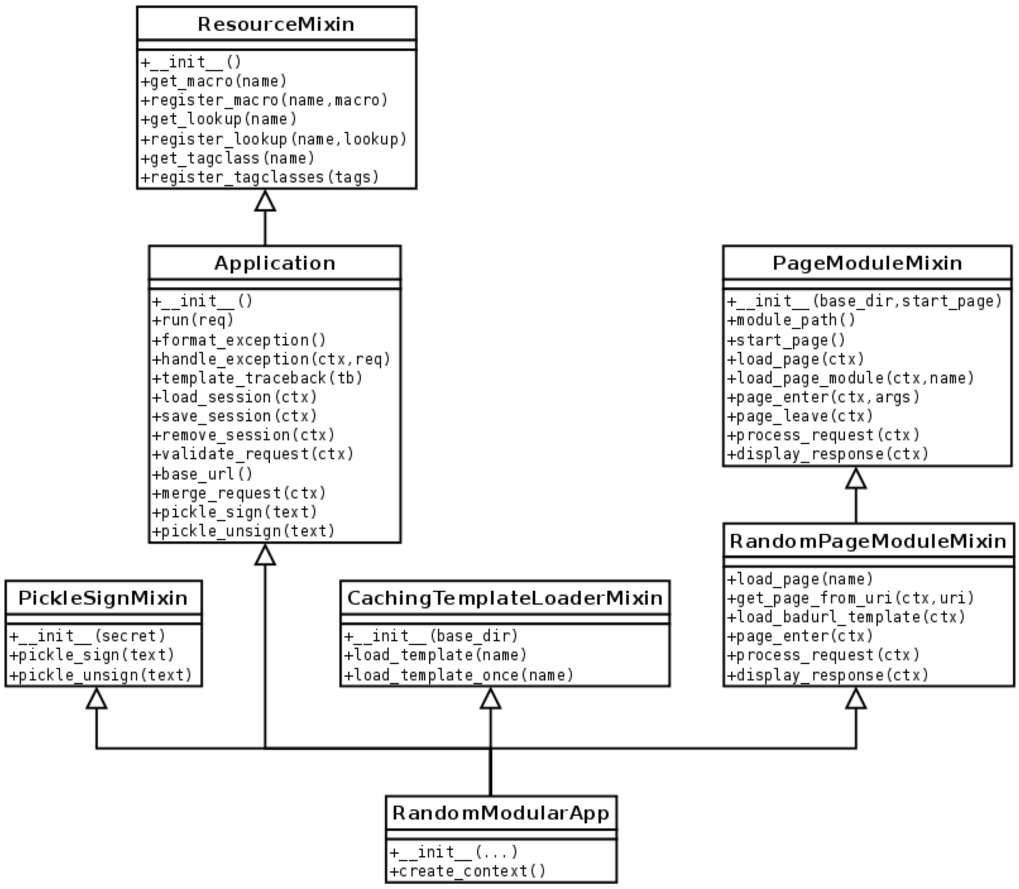


Figure 11.15: The `RandomModularApp` class

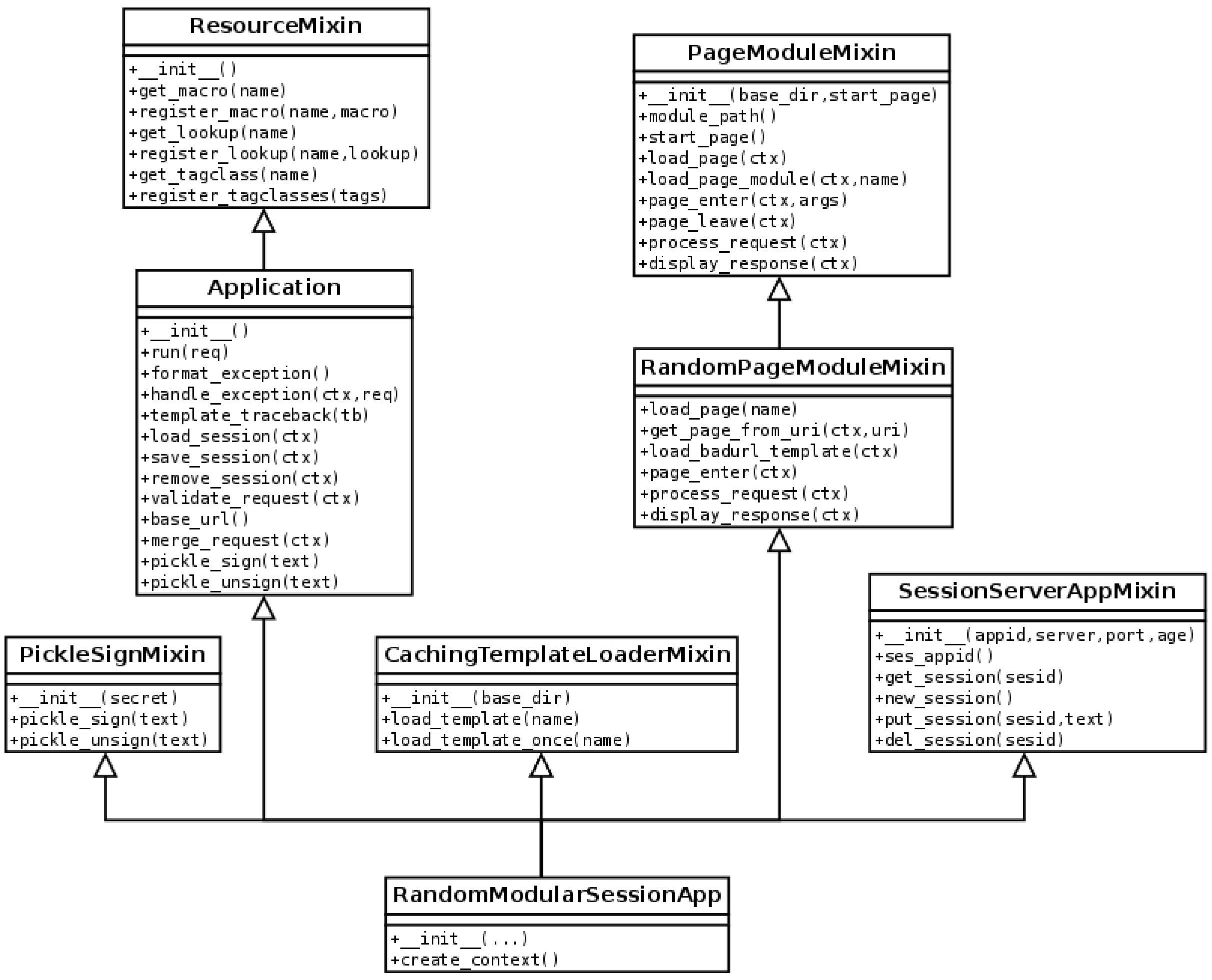


Figure 11.16: The `RandomModularSessionApp` class

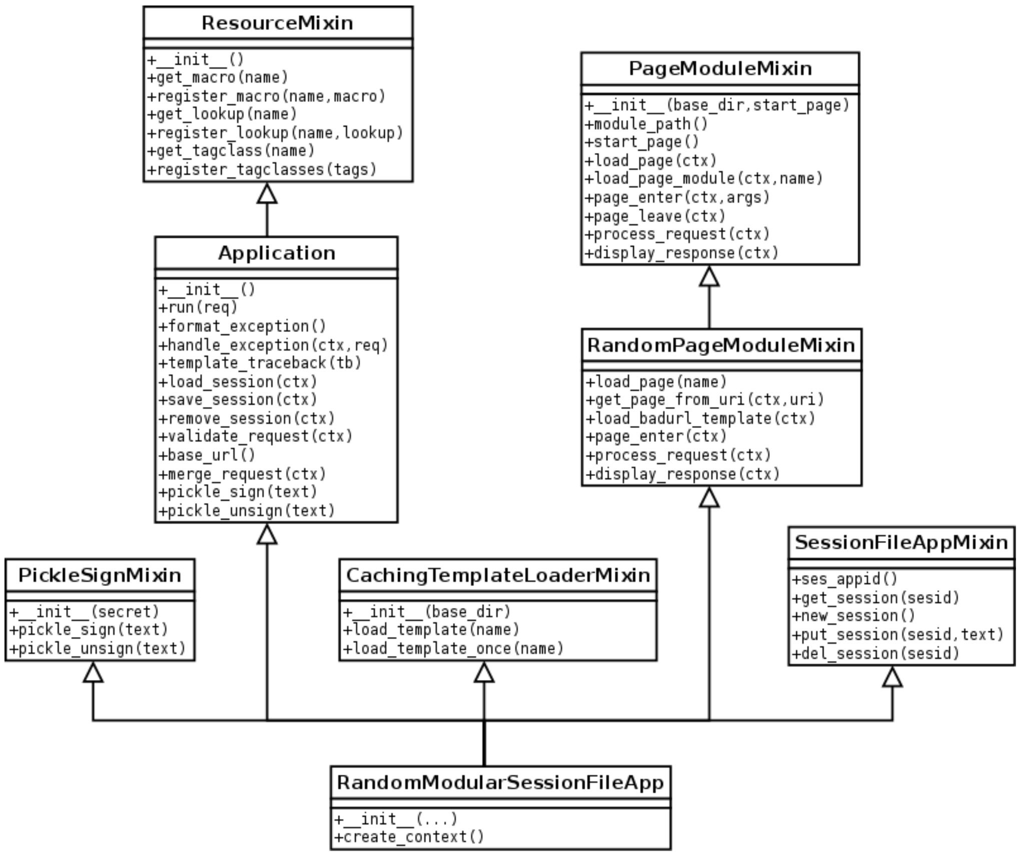


Figure 11.17: The `RandomModularSessionFileApp` class

CHAPTER

TWELVE

SUMMARY OF CHANGES

This chapter describes the changes made to Albatross between releases.

12.1 Release 1.40.

This section describes the changes made to Albatross since release 1.36.

12.1.1 Major Changes

- **Conversion of documentation to reStructuredText**

  With the Python documentation moving from LaTeX markup to reStructuredText and the Sphinx documentation generator, we could no longer rely on the documentation utilities released with Python. The Albatross documentation has been converted to ReST, and uses Sphinx to render this to HTML and PDF formats.

- **Albatross Forms**

  A simple HTML form generation and validation framework has been added to Albatross (as an optional extension under the module name `albatross.ext.form`).

12.1.2 Bug Fixes

- Redirects were not calling `req.return_code()` - for FastCGI deployment, this meant the request would hang [11164].
- The python built-in `__import__()` gained extra arguments with python 2.6 - the decode session import hook has been changed to pass all arguments (positional and keyword) [14823].
- When rendering tags to HTML, the handling of the noescape attribute was not consistently honoured. Attribute rendering has been moved to the common Tag base class as the `write_attrib()` method [15398].
- Relaxed tag recognition regular expressions, so that malformed tag attributes with no value, such as `<al-input name= />`, are still recognised as valid Albatross tags, albeit with a null name attribute (which subsequently generates an error) [15398].

12.1.3 Miscellaneous Changes

- Miscellaneous changes to RedHat, Ubuntu, Solaris and OS X packaging rules.
- Reorganised the unit tests to eliminate the use of explicitly assembled test suites, relying instead on the `unittest` module to collect methods of TestCase subclasses. The documentation example tests were also reimplemented as TestCase subclasses, allowing them to be run via common test driver.

12.2 Release 1.36

This section describes the changes in release 1.36 of Albatross that were made since release 1.35.

12.2.1 New Features

New <al-for vars="..."> attribute

A `vars` attribute has been added to <al-for> - this inserts the iterator value into the local namespace. The iterator value was previously only accessible via the iterator `value()` method.

<al-for> now accepts iterators

<al-for> now loops over sequences using the iterator protocol by default. The previous behaviour of indexing the sequence is retained where pagination or columns are requested.

Default arguments for macros using <al-setdefault>

New <al-setdefault> tag which allows macros to define default values for their arguments (suggested by Greg Bond).

<al-expand> argument shorthand

Macro expansion can now use attributes on the <al-expand> tag to specify macro arguments (suggested by Greg Bond).

Access to HTTP environment variables

Added get_param() method to Request classes, which deployment method agnostic access to http "environment" variables.

12.2.2 Functional Changes

Safer request merging

`NameRecorderMixin` no longer falls back to merging all request fields if no `__albform__` field is present. Applications using the `NameRecorderMixin` and GET-style requests will need to explicitly merge relevent fields with the `ctx.merge_vars(...)` method.

Switch from MD5 to SHA1

Pickles embedded in hidden fields are now signed using the HMAC-SHA1 algorithm (was MD5).

New FastCGI driver now default

The experimental FastCGI driver included in version 1.35 now becomes the default FastCGI driver. The old driver has been renamed fcgiappold. The new driver implements the FastCGI protocol itself, rather than depending on an external module.

12.2.3 Bug Fixes

<al-option> fixes

If the <al-option> tag was not immediately enclosed within an <al-select> tag, it would be silently ignored. <al-option> now works within containing flow control tags such as <al-for> or <al-if>, has improved error reporting, and supports attributes for dynamically setting value and label.

Session server client code enhancements

Communication with the session server has been made more robust. EINTR is handled, as are partial reads and writes, and requests are restarted if the socket closes.

Fix to input type recording

Mixing radio inputs with other inputs of the same name did not raise an exception (reported by Michael Neel).

Fixed incorrect constant in FastCGI driver

FCGI_MAX_DATA was incorrect due to an operator precedence mistake. Found by Tom Limoncelli.

12.2.4 Miscellaneous Changes

Clean up pre-python 2.2 constructs

Cleaned up and replaced many pre-python 2.2 constructs.

New and updated unit tests

Many new tests have been added, and existing tests restructured.

12.3 Release 1.35

This section describes the changes in release 1.35 of Albatross that were made since release 1.34. Note that release 1.34 was an internal release.

12.3.1 New Features

New FastCGI module

A drop-in replacement for the `fcgiapp` module, called `fcgiappnew` has been added. This version implements the FastCGI protocol itself, rather than relying on an external module to implement the protocol (we have not been able to clarify the license of the fcgi.py module). This new module addresses several minor problems with fcgi.py, and should be faster, although it should not be used in critical applications until it has received wider testing. This module will eventually be renamed to replace fcgiapp (at which point, the fcgiappnew name will dropped).

12.3.2 Functional Changes

Validate extension tag names

When extension tags (alx-*) are registered, their name is now checked against the template parsing regexp to ensure they can subsequently be matched.

12.3.3 Bug Fixes

Enhance AnyTag with knowledge of empty HTML tags

The AnyTag functionality was given knowledge of HTML tags for which the close tag is forbidden, so it can avoid generating XHTML empty tag (which could cause the page to fail HTML validation).

Input tags with disabledbool attribute

When the disabledbool attribute was used on input tags, the disabled state was not being passed through to the input registry within the `NameRecorderMixin`.

Improve session server handling of aborted connections

If a client closed it's connection to the session server while the server had data pending for the client, a subsequent del_write_file would generate an exception, killing the session server.

12.4 Release 1.33

This section describes the changes in release 1.33 of Albatross that were made since release 1.32.

12.4.1 Bug Fixes

ctx.set_value()

Fixed handling of tree iterator backdoor and improved error reporting.

12.5 Release 1.32

This section describes the changes in release 1.32 of Albatross that were made since release 1.31.

12.5.1 Bug Fixes

_caller_globals()

To obtain a reference to the current frame, _caller_globals was raising and catching an exception, then extracting the tb_frame member of sys.exc_traceback. sys.exc_traceback was deprecated in python 1.5 as it is not thread-safe. It now appears to be unreliable in 2.4, so _caller_globals has been changed to use sys._getframe().

ctx.set_page() from start page

If ctx.set_page() was called from within the start page, then the wrong page methods (page_enter, page_display, etc) would be called (those of the initial page, rather than the page requested via set_page).

12.6 Release 1.31

This section describes the changes in release 1.31 of Albatross that were made since release 1.30.

12.6.1 Bug Fixes

RandomPage error handling

Fixes to handling of missing RandomPage page modules.

12.7 Release 1.30

This section describes the changes in release 1.30 of Albatross that were made since release 1.20.

12.7.1 Functional Changes

Evaluate any attribute of any tag

Arbitrary HTML tags can now access the templating engine by prefixing the tag with “al-“. Attributes of the tag can then be selectively evaluated to derive their value. Appending “expr” to the attribute name causes the result of evaluating the expression to be substituted for the value of the attribute. Appending “bool” results in the attribute value being evaluated in a boolean context, and if true, a boolean HTML attribute is emitted. For example:

```
<al-td colspanexpr="i.span()">
```

could produce

```
<td colspan="3">
```

and:

```
<al-input name="abc.value" disabledbool="abc.isdisabled()">
```

could produce

```
<input name="abc.value" disabled>
```

Enforce only one definition of macros and lookups

Since macros and lookups are an application global resource, they can only be defined once per application, however this was not previously enforced. Redefinition of macros or lookups will now result in an ApplicationError exception.

In-line expansion of `<al-lookup>`

The `<al-lookup>` tag can now be optionally expanded in place. If the tag has an expr= attribute, this will be evaluated and used as the value to look up, and the results of the lookup substituted for the tag.

Functionality of named lookups remains unchanged.

New `<al-require>` tag

A new <al-require> tag has been added to allow templates to assert that specific Albatross features are available, or templating scheme version number is high enough. For instance, the addition of the "Any Tag" functionality has resulting in the templating version incrementing from 1 to 2.

Set Cache-Control header

`Cache-Control:  no-cache` is now set in addition to `Pragma:  no-cache`.

`Cache-Control` was introduced in HTTP/1.1, prior to this the same effect was achieved with `Pragma`. Some browsers change their behaviour depending on whether the page was delivered via HTTP/1.1 or HTTP/1.0.

Simplified Session Cookie handling

Session cookie handling has been simplified.

12.7.2 Bug Fixes

FastCGI finalisation

FastCGI apps were not being explicitly finalised, relying instead on their object destructor, with the result that writing application output (or errors) would be indefinitely deferred if object cycles existed. We now call `fcgi.Finish()` from the fcgiapp `Request.return_code()` method.

Delete traceback objects

When handling exceptions, the traceback is now explicitly deleted from the local namespace to prevent cycles (otherwise the garbage collection of other objects in the local namespace will be delayed).

`<al-select>` fixes

Two fixes to the `<al-select>` tag: the albatross-specific "list" attribute was leaking into resulting markup, and the use of the "expr" attribute would result in invalid markup being emitted.

Illegal placement of `<input>` tag

Thanks to Robert Fendt for picking this up: the Albatross-generated hidden field input element must not appear naked inside a form element for strict HTML conformance. The solution is to wrap the input elements in div.

Allow BranchingSessions to be deleted

BranchingSession sessions could not be "deleted" because each interaction is a separate session. The solution implemented is to add a dummy "session" shared by all branches, which is deleted when one branch "logs out".

12.8 Release 1.20

This section describes the changes in release 1.20 of Albatross that were made since release 1.11.

12.8.1 Functional Changes

New `BranchingSessionContext`

A persistent problem with server-side sessions is the browser state getting out of synchronisation with the application state. This occurs when the user reloads the page or uses the “back” button.

A new `BranchingSessionContext` application context class has been added that attempts to work around this problem by creating a new server-side session for every interaction with the browser. The unique session identifier is stored in a hidden form field, rather than a cookie.

The new Context class is intended to be used with the server-side Application classes, and provides a similar experience to storing the context in a hidden form field, without the overhead and security issues of sending the context on a round-trip through the user’s browser.

No effort is made at this time to control the resources used by these server- side sessions, other than expiring them after `session_age` seconds.

Improved Request classes

The Request classes provide the interface between specific application deployment models (CGI, FastCGI, mod_python, etc), and the Albatross application. These classes have been refactored to extract common functionality into a new RequestBase class. The Request classes also now have methods for passing status back to browser.

Page Module loading

The page module loader in PageModuleMixin has been reimplemented so that it does not pollute `sys.modules`. Page modules are now loaded into a synthetic module’s namespace, rather than the global module namespace. This will break code that defined classes in page modules and placed instances of those classes into the session.

Multi-instance response headers now supported

Some HTTP headers can appear multiple times (for example Set-Cookie) - the response handling has been modified to allow multiple instances of a header. `ResponseMixin.get_header()` now returns a list of strings, rather than just a string. The httpdapp module has also been updated to allow multiple instances of a header, keeping headers in a list rather than a dictionary.

simpler `req_equals()` matching with image maps

`ctx.req_equals(name)` now checks for `name.x` if `name` is not found. This makes using image maps as buttons easier (from Michael C. Neel).

12.8.2 Bug Fixes

`redirect_url()` fixes

Under some circumstances, `redirect_url()` would redirect to incorrect or invalid URLs (for example, an https app would redirect to http) - the URI parsing has been refactored, and this bug has been fixed. Tests were also added for the refactored URI parsing.

Improved request status handling

1. Symbolic names are now defined for the RFC1945 status header values, such as HTTP_OK, HTTP_MOVED_PERMANENTLY, HTTP_MOVED_TEMPORARILY and HTTP_INTERNAL_SERVER_ERROR
2. The Request classes (deployment model adaptors) and Application `run()` method have been updated to correctly pass the returned status back to the client.

Response header matching now case-insensitive

Response header names were being matched in a case-sensitive way - this was incorrect and has been fixed.

Cookie handling fixes

1. A Cookie path bug was noticed when Albatross applications were used with the Safari browser. `absolute_base_url()` was generating a trailing slash on the returned application URL (so /path/app.cgi/ instead of /path/app.cgi). This was causing problems for requests like /path/app.cgi?blah in that Safari did not send the cookie (probably correctly).
2. When an application was accessed via https, the `secure` attribute on any resulting cookies was not being set. This attribute marks the cookie to be only returned via an https connection. The `secure` attribute is now set.
3. Cookie max-age was being allowed to default - this is now explicitly set to match the configured session age (from the Application `session_age` parameter).

12.9 Release 1.11

This section describes the changes in release 1.11 of Albatross that were made since release 1.10.

12.9.1 Functional Changes

`<al-select>/<al-input>` consistency

`<al-select>` handling of `name`, `expr`, `valueexpr` and `value` attributes has been made consistent with that of `<al-input>`.

`absolute_base_url` method

New method `absolute_base_url()` has been added to the `AppContext`.

`al-httpd` enhancements

Matt Goodall has continued to improve the capabilities of the `al-httpd` program and `httpdapp.py` so that it is now possible to run all of the CGI based sample applications.

You can now initialise the `static_resources` from the command line. For example, the tree samples can be executed to serve up their images like this:

```
$ cd samples/tree2
$ al-httpd tree.app 8080 /alsamp/images ../images/
```

XHTML fixes

The `<al-input>` and `<al-img>` tags now output XHTML compliant end tags.

12.9.2 Bug Fixes

`mod_python` support

Greg Bond fixed a `cgiapp` field handling incompatibility with `mod_python` 3.

`get_servername()` support

The `get_servername()` method of the `cgiapp` and `fcgiapp` `Request` classes now use the `HTTP_HOST` environment variable rather than `SERVER_NAME`.

Multiple cookies

All session cookies now include a path attribute. This prevents multiple redundant cookies being set for all URI paths in an application.

12.10 Release 1.10

This section describes the changes in release 1.10 of Albatross that were made since release 1.01.

12.10.1 Functional Changes

FastCGI support

Matt Goodall developed support for deployment of applications via FastCGI. FastCGI applications import their `Request` class from `albatross.fcgiapp`.

Standalone BaseHTTPServer support

Matt Goodall developed support for standalone deployment of applications via the standard Python `BaseHTTPServer` module. The `al-httpd` program can be used to deploy a CGI application as a standalone `BaseHTTPServer` server.

Exception Classes

All Albatross exceptions have been redefined to indicate the source of the error; user (`UserError`), programmer (`ApplicationError`), or Albatross itself (`InternalError`). The `ServerError` exception reports problems related to the session server, `SecurityError` reports either a programmer error, or a user attempt to access restricted values in the execution context, and `TemplateLoadError` reports failures to load templates.

All of the exceptions inherit from `AlbatrossError`.

The `albatross.common` module defines the exceptions.

Response Header Management

All response header management has been moved to the `ResponseMixin` class. The `AppContext` class now inherits from `ResponseMixin`. The `Request` class no longer tracks whether or not headers have been sent.

`ResponseMixin` provides the ability to set, get, delete, and send headers. Headers are automatically sent when the application sends any content to the browser.

The `write_headers()` method has been deleted from the following classes; `SimpleAppContext`, `SessionAppContext`, `SessionServerContextMixin`, `SessionFileContextMixin`, `SessionFileAppContext`.

For `SessionServerContextMixin` and `SessionFileContextMixin` the `Set-Cookie` header is set when session is created or loaded.

HTTP Response Codes

The `Application.run()` method no longer unconditionally returns an HTTP response code of `200`. The returned response code is retrieved from the `Request.status()` method. You can call the `Request.set_status()` method to override the default HTTP response code of `200`.

File Uploading

The `<al-input>` tag now supports `type="file"` input tags. When you use file input tags the enclosing `<al-form>` tag automatically adds the `enctype="multipart/form-data"` attribute to the generated `<form>` tag.

The `albatross.cgiapp` and `albatross.apacheapp` modules define a `FileField` class which provides access to uploaded files. During request merging the `Request.field_file()` method returns instances of `FileField` for uploaded files.

Session Changes

The `Application.run()` method now saves the session before flushing the response to the browser. This allows applications to support dynamically generated images.

The `SessionBase.add_session_vars()` method now raises an `ApplicationError` exception if you attempt to add variables to the session which do not exist in the local namespace.

The `SessionBase.default_session_var()` method allows you to add a variable to the session and place it in the local namespace at the same time.

Session saving previously silently removed session values which could not be pickled. Now unpickleable values are reported via an `ApplicationError` exception.

Errors handling and reporting during session loading has been improved.

Exception Formatting and Handling

The exception formatting in `Application.handle_exception()` has been moved into the `format_exception()` method. Applications can perform their own exception formatting and/or send formatted exceptions to locations other than the browser.

Unicode

The `ExecuteMixin.write_content()` method now converts unicode to UTF-8.

Execution Context Available to Template Expressions

During `NamespaceMixin.eval_expr()` the execution context is temporary placed into the local namespace as the variable *__ctx__*.

Request Merging

`NamespaceMixin.set_value()` ignores attempts to set Albatross iterators that are not present in the namespace.

`NamespaceMixin.set_value()` produces a nice syntax error like report when an illegal field name is used.

Locating Globals for Template Expressions

The `_caller_globals()` function has been changed to use the name of a function rather a stack frame count. This is used by the methods `AppContext.run_template()`, `AppContext.run_template_once()`, `RandomPageModuleMixin.display_response()`, and `SimpleContext.__init__()` to locate the module whose globals will be used as the global namespace for evaluating template expressions.

Tree Handling

The `<al-tree>` tag now has a `single` attribute which enables the single select mode. In single select mode, selecting a node automatically deselects any previously selected node.

The `<al-input>` tag now supports the `treefold="expr"`, `treeselect="expr"`, and `treeellipsis="expr"` attributes. The expression specifies a tree node that is used to construct a tree iterator input value.

The following methods have been added to the `LazyTreeIterator` class; `load_children()`, `is_selected()`, `select_alias()`, `open_alias()`.

Lookup Evaluation

The `<al-lookup>` tag now registers the lookup during template parsing rather than during evaluation. This allows template code to make use of lookups that are defined later in the same file. The item dictionary is created the first time that the lookup is used rather than when the template is interpreted.

Macro Argument Evaluation

Macro arguments are now evaluated when they are referenced rather than before they are passed to a macro.

This removes a limitation where you could not define macros including `<al-form>` tags that retrieved `<al-input>` tags from their arguments. Previously the `<al-input>` tags passed as macro arguments would have been evaluated outside of the context of the form defined in the macro. This effectively made the input tags invisible to the form recorder.

`<select>/<option>` formatting

The `<al-option>` and `<al-select>` tags always write `</option>` close tags.

`noescape` tag attribute

The `noescape` attribute has been added to the `<al-input>`, `<al-img>`, `<al-select>`, and `<al-textarea>` tags.

Documentation

The templates reference documentation has been completely restructured to improve clarity. All attributes of each tag have been documented.

12.10.2 Bug Fixes

Lingering Content Trap

In some circumstances exceptions would leave a content trap in place that prevented an error report from being written to the browser.

Session Loading

More exceptions are trapped by the session unpickling code to make error handling more robust.

Random Page Module Loading

Fixed a bug where an import error inside a page module loaded by the `RandomPageModuleMixin` was being handled as if page module could not be located.

Session Id Cookie Handling

Fixed cookie handling which previously could not cope with missing session id when the cookie was present.

`<al-input>` tag

Fixed bug in checkbox.

Only prevent the generation of the `value` attribute when the value is `None`.

`<al-input>` tag

The `apacheapp Request` class now works for mod_python 2.3 and 3.0.

INDEX