Webware for Python 3

Release 3.0.10

Christoph Zwerschke et al.

May 27, 2023

CONTENTS:

CHAPTER ONE

OVERVIEW

1.1 Synopsis

Webware for Python is a framework for developing object-oriented, web-based applications.

The project had been initiated in 1999 by Chuck Esterbrook with the goal of creating the ultimate web development suite for Python, and it soon got a lot of attraction. Jay Love, Geoff Talvola and Ian Bicking were other early contributors and core developers.

They created a mature and stable web framework that has been used in production for a long time by a variety of people in different environments. Since then, a lot of other web frameworks for Python emerged and have taken the lead, such as Django, Flask or Pyramid, while Webware for Python got less attention. Webware for Python was still available, maintained and slightly improved by Christoph Zwerschke, and happily used here and there, but did not change much over the years.

Since Webware for Python was based on Python 2, for which support ended 20 years later at the end of 2019, but there were still Webware for Python applications in the wild running happily after 2020, Christoph Zwerschke created a Python 3 based edition of the project called Webware for Python 3.

1.2 Design Points and Changes

Webware for Python 3 kept the following ideas and key goals from the original project:

- **Simplicity**. Webware's code is quite simple and easy to understand if you feel the need to extend it.
- **Servlets**. Similar to Java servlets, they provide a familiar basis for the construction of web applications.
- **Robustness**. A crash of one page will not crash the server. Exception reports are logged and easy to read when developing.
- **Object-programming programming** (making good use of multiple inheritance and the template method pattern).
- **Extensibility** via plug-ins.
- **Python Server Pages** (PSP, similar to ASP, PHP and JSP) as a built-in plug-in.
- Built-in plug-ins for **Task scheduling** and **User management**.
- Excellent **documentation** and numerous **examples**.

Another key goal of the original project was to provide a "Pythonic" API, instead of simply copying Java APIs. However, the project was created when Python 2 was still in its infancy, lacking many modern features and conventions such as PEP-8. Therefore, the Webware for Python API is a bit different from what is considered "Pythonic" nowadays. Particularly, it uses getters and setters instead of properties (but without the "get" prefix for getters), and camelCase method names instead of snake_case. In order to facilitate migration of existing projects, Webware for Python 3 kept this old API, even though it is not in line with PEP-8 and could be simplified by using properties. Modernizing the API will be a goal for a possible third edition of Webware for Python, as well as using the Python logging facility which did not yet exist when Webware for Python was created and is still done via printing to the standard output.

The plug-in architecture has also been kept in Webware for Python 3, but now implemented in a more modern way using entry points for discovering plug-ins. Old plug-ins are not compatible, but can be adapted quite easily. The old Webware for Python installer has been replaced by a standard setup.py based installation.

The most incisive change in Webware for Python 3 is the discontinuation of the threaded application server that was part of the built-in "WebKit" plug-in and actually one of the strong-points of Webware for Python. However, a threaded application based architecture may not be the best option anymore for Python in the age of multi-core processors due to the global interpreter lock (GIL), and maintaining the application server based architecture would have also meant to maintain the various adapters such as `mod_webkit` and the start scripts for the application server for various operating systems. This did not appear to be feasible. At the same time, Python nowadays already provides a standardized way for web frameworks to deploy web applications with the Python Web Server Gateway Interface (WSGI). By making the already existing Application class of Webware for Python usable as a WSGI application object, Webware applications can now be deployed in a standardized way using any WSGI compliant web server, and the necessity for operating as an application server itself has been removed. Webware for Python 3 applications deployed using `mod_wsgi` are even performing better and can be scaled in more ways than applications for the original Webware for Python that have been deployed using `mod_webkit` which used to be the deployment option with the best performance. During development, the waitress WSGI server is used to serve the application, replacing the old built-in HTTP server. As a structural simplification that goes along with the removal of the WebKit application server, the contents of the WebKit plug-in are now available at the top level of Webware for Python 3, and WebKit ceased to exist as a separate plug-in.

The second incisive change in Webware for Python 3 is the removal of the "MiddleKit" as a built-in plug-in. This plug-in served as a middle tier between the data storage and the web interface, something that nowadays is usually done with an object relational mapper (ORM) such as SQLAlchemy. MiddleKit was a powerful component that many users liked and used in production, but was also pretty complex, with adapters to various databases, and therefore hard to maintain. It made sense to swap it out and provide MiddleKit for Webware for Python 3 as a separate, external plug-in on GitHub. Also removed were the "CGIWrapper", "COMKit" and "KidKit" built-in plug-ins, because they have become obsolete or outdated. The other built-in plug-ins were less complex than MiddleKit and were kept as built-in plug-ins of Webware for Python 3. Particularly, "PSP, "UserKit" and "TaskKit" are still available in Webware for Python 3.

To facilitate web development with Webware for Python 3, a `webware` console script has been added that can be used to create working directories for new application and start the development server. This script replaces the old `MakeAppWorkDir` and `AppServer` scripts. When creating a new working directory, a WSGI script will also be created that can be used to attach the application to a web server.

The documentation contexts of the various plug-ins have been replaced by a common Sphinx based documentation provided in the top-level `docs` directory. The tests are still contained in `Tests` subdirectories at the top and plug-in levels, but the test suite has been expanded and is using the unittest framework consistently. The twill tests have also been replaced by unit tests based using WebTest. They make sure that all servlets in the examples and testing contexts work as expected. Since Webware for Python 3 uses WSGI, WebTest can now also be used to test applications built with Webware for Python 3.

Otherwise, not much has been changed, so that migrating existing Webware for Python applications to Webware for Python 3 should be straight forward. Of course, you still need to migrate your Webware applications from Python 2 to Python 3, but meanwhile a lot of tools and guidelines have been provided that help making this process as painless as possible.

See the *List of Changes* and the *Migration Guide* for more detailed information.

1.3 Download and Installation

See the chapter on *Installation* for instructions how to download and install Webware for Python 3.

1.4 Documentation

This documentation is available online via GitHub Pages and via Read the Docs.

1.5 Feedback, Contributing and Support

You can report issues and send in pull requests using the GitHub project page of Webware for Python 3. If you want to be notified when new releases are available, you can use the “Watch” feature of GitHub.

CHAPTER

TWO

INSTALLATION

2.1 Python Version

Webware for Python 3 requires at least Python version 3.6.

2.2 Create a Virtual Environment

Though you can install Webware for Python 3 into your global Python environment, we recommend creating a separate virtual environment for every Webware for Python 3 project.

To create such a virtual environment in the `.venv` subdirectory of the current directory, run the following command:

```
python3 -m venv .venv
```

If you are using Windows, may may need to run the following instead:

```
py -3 -m venv .venv
```

2.3 Activate the Virtual Environment

To activate the virtual environment, you need to execute the "activate" command of the virtual environment like this:

```
. .venv/bin/activate
```

Or, if your are using Windows, the "activate" command can be executed like this:

```
.venv\Scripts\activate
```

2.4 Installation with Pip

With the virtual environment activated, you can now download and install Webware for Python 3 in one step using the following command:

```
pip install "Webware-for-Python>=3"
```

For developing with Webware for Python, you will probably also install "extras" as explained below.

2.5 Installing "Extras"

When installing Webware for Python 3, the following "extras" can optionally be installed as well:

- "dev": extras for developing Webware applications
- "examples": extras for running all Webware examples
- "test": extras needed to test all functions of Webware
- "docs": extras needed to build this documentation

On your development machine, we recommend installing the full "test" environment which also includes the other two environments. To do that, you need to specify the "Extras" name in square brackets when installing Webware for Python 3:

```
pip install "Webware-for-Python[dev]>=3"
```

2.6 Installation from Source

Alternatively, you can also download Webware for Python 3 from PyPI, and run the `setup.py` command in the tar.gz archive like this:

```
setup.py install
```

You will then have to also install the "extra" requirements manually, though. Have a look at the setup.py file to see the list of required packages.

2.7 Check the Installed Version

In order to check that Webware has been installed properly, run the command line tool `webware` with the `--version` option:

```
webware --version
```

This should show the version of Webware for Python 3 that you have installed. Keep in mind that the virtual environment into which you installed Webware for Python 3 needs to be activated before you run the "webware" command.

CHAPTER

THREE

LIST OF CHANGES

3.1 What's new in Webware for Python 3

This is the full list of changes in Webware for Python 3 (first version 3.0.0) compared with Webware for Python 2 (last version 1.2.3):

- Webware for Python 3 now requires Python 3.6 or newer, and makes internal use of newer Python features where applicable. Webware applications must now be migrated to or written for Python 3.
- The "Application" instance is now callable and usable as a WSGI application.
- The application server ("AppServer" class and subclasses including the "ThreadedAppServer") and the various adapters and start scripts and other related scripts for the application server are not supported anymore. Instead, Webware applications are now supposed to be served as WSGI applications using a WSGI server such as waitress, which is now used as the development server.
- The "ASSStreamOut" class has been replaced by a "WSGIStreamOut" class. The "Message" class has been removed, since it was not really used for anything, simplifying the class hierarchy a bit.
- The Application now has a development flag that can be checked to modify the application and its configuration depending on whether it is running in development or production mode.
- The custom "install" script has been replaced by a standard "setup" script, Webware for Python 3 is now distributed as a normal Python package that can be installed in the usual way. The "ReleaseHelper" and "setversion" scripts are not used anymore.
- The "MakeAppWorkDir" script has been moved to a new "Scripts" directory, which now also contains a "WSGIScript" and a "WaitressServer" script which serve as replacements for the old "AppServer" and "Launch" start scripts. These scripts can now be started as subcommands of a new webware console script, which serves as a new common Webware CLI.
- Instead of the "AutoReloadingAppServer", you can use the "reload" option of the WaitressServer script which uses hupper to monitor the application files and reload the waitress server if necessary. The "ImportSpy" has been removed.
- The classes of the core "WebKit" component are now available at the root level of Webware for Python 3, and the WebKit component ceased to exist as a separate plug-in.
- Some built-in plug-ins are not supported anymore: "CGIWrapper", "ComKit" and "KidKit".
- "MiddleKit" is not a built-in plug-in anymore, but is provided as a separate project on GitHub now (WebwareForPython/w4py3-middlekit).
- Webware now uses entry points for discovering plug-ins instead of the old plug-in system, and the plug-in API has slightly changed. Existing plug-ins must be adapted to Python 3 and the new plug-in API.

- The documentation has been moved to a separate directory and is built using Sphinx, instead providing a “Docs” context for Webware and every plug-in, and using custom documentation builders in the install script. The existing content has been reformatted for Sphinx, adapted and supplemented.
- The various examples have been slightly improved and updated. Demo servlets showing the use of Dominate and Yattag for creating HTML in a Pythonic way have been added.
- The side bar page layout now uses divs instead of tables.
- The test suite has been expanded and fully automated using the unit testing framework in the Python standard library. We also use tox to automate various checks and running the test suite with different Python versions.
- In particular, end-to-end tests using Twill have been replaced by more efficient unit tests using WebTest.
- Internal assert statements have been removed or replaced with checks raising real errors.
- The style guide has been slightly revised. We now rely on flake8 and pylint instead of using the custom “checksrc” script.

See also the list of releases on GitHub for all changes in newer releases of Webware for Python 3 since the first alpha release 3.0.0a0.

CHAPTER

FOUR

MIGRATION GUIDE

In this chapter we try to give some advice on what needs to be done to migrate an existing Webware for Python application to Webware for Python 3.

Regarding the API, we tried to stay compatible with Webware for Python 2 as much as possible, even though modern Python uses different naming conventions and prefers the use of properties over getter and setter methods. So, in this regard, we expect a migration to Webware for Python 3 to be very smooth. The main points in a migration will be the conversion of the application from Python 2 to Python 3. the adaptation to the use of the WSGI standard instead of the custom application server, and maybe the usage of Webware plug-ins that are not supported anymore and may need to be migrated as well.

4.1 Check which Webware plug-ins you were using

First you should check whether the plug-ins your application is using are still available as built-ins plugin of Webware for Python 3 (w4py3) or as externally provided plug-ins. PSP is still provided as a built-in plug-in. MiddleKit is now provided as an external plug-in on GitHub (w4py3-middlekit). The "COMKit", "CGIWrapper" and "KidKit" built-in plug-ins have been discontinued. Other external plug-ins that have been developed for Webware for Python 2 must first be ported to Webware for Python 3 before you can use them. See the section on *Plug-ins* for details on how to write plug-ins for Webware for Python 3.

4.2 Migrate your application to Python 3

The main migration effort will be porting your Webware application from Python 2 to Python 3. More precisely, Webware for Python 3 requires Python 3.6 or newer. This effort is necessary anyway, if you want to keep your Webware application alive for some more years, because the Python foundation declared to end Python 2 support on January 1st 2020, which means that Python 2 will also not be supported by newer operating systems anymore and not even get security updates anymore. The positive aspect of this is that your Webware application will run slightly faster and you can now make use of all the modern Python features and libraries in your application. Particularly, f-strings can be very handy when creating Webware applications.

We will not go into the details of migrating your application from Python 2 to Python 3 here, since much good advice is already available on the Internet, for instance:

- Porting Python 2 Code to Python 3 (Brett Cannon)
- Supporting Python 3: An in-depth guide (Lennart Regebro)
- The Conservative Python 3 Porting Guide (Peter Viktorin et al)
- How To Port Python 2 Code to Python 3 (Lisa Tagliaferri)
- Migrating from Python 2 to Python 3 (Nick Heath)

- Migrating Applications From Python 2 to Python 3 (Mahdi Yusuf)

Note that some of the how-tos also explain how to create code that is backward compatible with Python 2, which is much more difficult than just porting to Python 3, as we can do when migrating a Webware application to Python 3. So don't be frightened by the seeming complexity of the task – it is probably much easier than you may think.

One of the biggest problems when migrating a Python 2 application to Python 3 is often the fact that in Python 3, strings are now always Unicode, while in Python 2, the native strings were byte strings and you had to add a "u" prefix to indicate that a string should be Unicode. However, this should not be a big issue when converting a Webware for Python application. Code such as `self.write('<p>Hello, World!</p>')` does not need to be modified. The fact that the string that is written to the output stream had been a byte string in Python 2 and is a Unicode string now in Python 3 is a detail that you as the application developer do not need to care about. Webware for Python 3 will encode everything properly to UTF-8 for you behind the scenes. If necessary, you can also change the output encoding from UTF-8 to something else with the `OutputEncoding` setting in the application configuration, but nowadays, UTF-8 is the usual and normally best choice.

Traditionally, Webware applications used simple print statements to output error or debug messages for logging or debugging purposes. You will need to change these print statements with print function calls when migrating from Python 2 to Python 3. In a future version of Webware for Python, we may change this and support a proper logging mechanism instead.

4.3 Use a WSGI server instead of the WebKit application server

The other big change is that instead of using the custom "WebKit" application server, Webware for Python 3 utilizes the WSGI standard as the only way of serving applications. You will need to adapt your deployment accordingly. See the section on *Deployment* for instructions on how to get your application into production using WSGI.

Search your application for direct references to the `AppServer` instance which does not exist anymore in Webware for Python 3. In most cases, you can replace these with references to the `Application` instance which also serves as the WSGI callable.

Also, search for references to the former `WebKit` package. This package does not exist anymore as separate plug-in in Webware for Python 3, its classes can now be found directly in the top level package of Webware for Python 3. So an import statement like `from WebKit.Page import Page` should be changed to a simple `from Page import Page`.

CHAPTER

FIVE

COPYRIGHT AND LICENSE

5.1 The Gist

Webware for Python is open source, but there is no requirement that products developed with or derivative to Webware become open source.

Webware for Python is copyrighted, but you can freely use and copy it as long as you don't change or remove this copyright notice. The license is a clone of the MIT license.

There is no warranty of any kind. Use at your own risk.

Read this entire document for complete, legal details.

5.2 Copyright

CHAPTER

SIX

QUICKSTART

In this chapter we will show how you can create and run a “hello world” application with Webware for Python 3, and try out the example servlets provided with Webware. In the next chapter, we will then go into a little bit more detail and create a slightly more complex application.

We assume you have already installed Webware for Python 3 into a virtual environment as explained in the chapter on *Installation*.

6.1 The Webware CLI

Webware for Python 3 comes with a command line tool named “webware” that helps creating a new projects and starting a development web server. You can run `webware` with the `--help` option to see the available subcommands:

```
webware --help
```

Remember that you need to activate the virtual environment where you installed Webware for Python 3 first if you haven’t installed it globally.

6.2 Creating a Working Directory

You can use the subcommand `make` to start making a new Webware for Python application. This subcommand will create a new Webware for Python 3 application working directory with subdirectories for your Webware contexts, configuration, etc. and will also generate some boilerplate files to get you started. Again, use the `--help` option to see all available options:

```
webware make --help
```

Let’s start a simple “hello world” project using the `make` command:

```
webware make HelloWorld
```

You should see some output on the console explaining what has been created for your. Particularly, you should now have a subdirectory “HelloWorld” in your current directory.

Each Webware application consists of one ore more “contexts”, which correspond to different URL prefixes. Some contexts such as the “example” and “admin” contexts are already provided with Webware for Python by default.

6.3 Running the Webware Examples

You need to cd into the newly created application working directory first:

```
cd HelloWorld
```

Now you can run the application with the following command:

```
webware serve -b
```

The "serve" subcommand will start the development server, and the -b option will open your standard web browser with the base URL of the application – by default this is http://localhost:8080/. You should see a simple web page with the heading "Welcome to Webware for Python!" in your browser.

You should also a link to the "Exmples" context located at the URL http://localhost:8080/Examples/. This context features several example servlets which you can select in the navigation side bar on the left side. Note that some of the examples need additional packages which should be installed as "extras" as explained in the chapter on *Installation*.

Try one of the example now, e.g. the CountVisits servlet. This example demonstrates how to use Webware Sessions to keep application state. In the navigation bar you will also find a link to view the source of CountVisits. You will find that the source code for this servlet is very simple. It inherits from the ExamplePage servlet.

6.4 Using the Admin Context

Besides the "Examples" context, Webware for Python also provides an "Admin" context out of the box. By default it is located at the URL http://localhost:8080/Admin/. You will notice an error message that says you first need to add an admin password using the AdminPassword setting in the Application.config configuration file.

You will find the configuration file in the Configs subdirectory of your application working directory. The configuration file defines all settings for the running Webware application using standard Python syntax. Try changing the AdminPassword defined at the top. You will notice that if you reload the admin context in your browser, you will still see the message about a missing admin password. You need to restart the application in order to make the changed application configuration effective. To do so, stop the running Webware application in your console with the Ctrl-C key combination, and start it again.

You can use the -r option to automatically restart the development server when files used in your application are changed:

```
webware serve -r
```

The serve subcommands has a few more options which you can see by running it with the --help option:

```
webware serve --help
```

After adding an admin password, you should be able to log in to the admin context using the user name "admin" and the admin password you selected.

In the navigation bar on the left side you now see several admin servlets. For example, the "Plug-ins" servlet lists all the Webware for Python plug-ins currently installed.

6.5 A "Hello World" Example

The make subcommand also created a subdirectory MyContext that serves as the default context for your application. You can rename this context in the Application.config file, or give it a different name with the -c option of the make subcommand. Let's leave the name for now – since it is the default context, you do not need to pass it in the URL.

A newly created default context already contains one servlet Main.py. Again, you don't need to pass this name in the URL, because it is the default name for the directory index, defined in the DirectoryFile setting of the application configuration file.

Let's add another very simple servlet HelloWorld.py to our default context. But first, let's add a link to this servlet to the Main servlet. Open the MyContext/Main.py file in your editor and add the following line at the bottom of the method writeContent():

```
self.writeln('<ul><li><a href="HelloWorld">Hello, World!</a></li></ul>')
```

Everything that you pass to the writeln() method is written onto the current page. This is the main method you will be using in Webware for Python to create dynamic HTML pages. You can also use the write() method which does the same without appending a new-line character at the end. This approach is very similar to writing CGI applications. However, the servlets are kept in memory and not reloaded every time you visit a page, so Webware for Python is much more efficient. Also, servlets classes allow a much better structuring of your application using object oriented programming.

When you navigate to the start page of your application, you should now already see this link. For now, you will get an "Error 404" when trying to click on this link. In order to make it operational, save the following file as MyContext/HelloWorld.py in your application working directory:

```
from Page import Page

class HelloWorld(Page):

    def title(self):
        return 'Hello World Example'

    def writeContent(self):
        self.writeln('<h1>Hello, World!</h1>')
```

Now the link on the start page should work and you should see "Hello World!" on your page and whatever more you want to write in the *writeContent()* method above.

If you want to change the style of the page, use different colors or larger letters, you should use the writeStyleSheet() method to define an inline style sheet or link to a static CSS file. For example, try adding the following method to your *HelloWorld* class above:

```
    def writeStyleSheet(self):
        self.writeln('''<style>
h1 {
    color: blue;
    font-size: 40px;
    font-family: sans-serif;
    text-align: center;
}
</style>''')
```

CHAPTER

SEVEN

BEGINNER TUTORIAL

In this tutorial we will show how to write a very simple Webware application.

Again, we assume that you have Webware for Python 3 already installed in a virtual environment, and activate the virtual environment as explained in the chapter on *Installation*.

7.1 Creating a Working Directory

We'll first set up a directory dedicated to your application, the so-called "application working directory". Change into your home directory or wherever you want to create that working directory. We recommend creating the virtual environment and the application working directory as siblings in a dedicated base directory, which can be the home directory of a dedicated user that acts as "owner" of the application. Then run this command:

```
webware make -c Context -l Lib WebwareTest
```

You'll now have a directory "WebwareTest" in the current directory. Inside this directory will be several subdirectories and a couple files. The directories of interest are `Context` (that you specified with `-c context`) where you'll be putting your servlets; `Configs` that holds some configuration files; and `Lib` where you can put your non-servlet code.

For more information about the working directory and setting up the file structure for your application, see *Application Development*.

7.2 Changing the Webware Configuration

For the most part the configuration is fine, but we'll make a couple changes to make it easier to develop. For more information on configuration see the chapter on *Configuration*.

One change you may want to make to allow you to use more interesting URLs. In `Application.config`, change the `ExtraPathInfo` setting from `False` (the default) to `True`:

```
ExtraPathInfo = True
```

Otherwise the settings should be appropriate already for our purposes.

7.3 Creating and Understanding the Servlet

Webware's core concept for serving pages is the *servlet*. This is a class that creates a response given a request.

The core classes to understanding the servlet are `Servlet`, `HTTPServlet`, and `Page`. Also of interest would be the request (`Request` and `HTTPRequest`) and the response (`Response` and `HTTPResponse`) – the `HTTP-` versions of these classes are more interesting. There is also a `Transaction` object, which is solely a container for the request and response.

While there are several levels you can work on while creating your servlet, in this tutorial we will work solely with subclassing the `Page` class. This class defines a more high-level interface, appropriate for generating HTML (though it can be used with any content type). It also provides a number of convenience methods.

7.4 A Brief Introduction to the Servlet

Each servlet is a plain Python class. There is no Webware magic (except perhaps for the level one *import module based on URL* spell). *PSP* has more magic, but that's a topic for another chapter.

An extremely simple servlet might look like:

```
from Page import Page

class MyServlet(Page):

    def title(self):
        return 'My Sample Servlet'

    def writeContent(self):
        self.write('Hello world!')
```

This would be placed in a file `MyServlet.py`. Webware will create a pool of `MyServlet` instances, which will be reused. Servlets "write" the text of the response, like you see in the `writeContent()` method above.

Webware calls the servlet like this:

- An unused servlet is taken from the pool, or another servlet is created.
- `awake(transaction)` is called. This is a good place to set up data for your servlet. You can put information in instance variables for use later on. But be warned – those instance variables will hang around potentially for a long time if you don't delete them later (in `sleep`).
- Several low-level methods are called, which Page isolates you from. We will ignore these.
- `writeHTML()` is called. `Page` implements this just fine, but you can override it if you want total control, or if you want to output something other than HTML.
- `writeDocType()` would write something like `<!DOCTYPE html>`.
- The <head> section of the page is written. `title()` gives the title, and you probably want to override it.
- `writeStyleSheet()` is called, if you want to write that or anything else in the <head> section.
- The <body> tag is written. Have `htBodyArgs()` return anything you want in the <body> tag (like `onLoad="loadImages()"`).
- `writeBodyParts()` is called, which you may want to override if you want to create a template for other servlets.
- `writeContent()` should write the main content for the page. This is where you do most of your display work.

- The response is packaged up, the headers put on the front, cookies handled, and it's sent to the browser. This is all done for you.
- `sleep(transaction)` is called. This is where you should clean up anything you might have set up earlier – open files, open database connections, etc. Often it's empty. Note that `sleep()` is called even if an exception was raised at any point in the servlet processing, so it should (if necessary) check that each resource was in fact acquired before trying to release it.
- The servlet is placed back into the pool, to be used again. This only happens after the transaction is complete – the servlet won't get reused any earlier.

You only have to override the portions that you want to. It is not uncommon to only override the `writeContent()` method in a servlet, for instance.

You'll notice a file `Context/Main.py` in your working directory. You can look at it to get a feel for what a servlet might look like. (As an aside, a servlet called `Main` or `index` will be used analogous to the `index.html` file). You can look at it for a place to start experimenting, but here we'll work on developing an entire (small) application, introducing the other concepts as we go along.

7.5 A Photo Album

If you look online, you'll see a huge number of web applications available for an online photo album. The world needs one more!

You will need the Pillow library installed for this example. If you installed Webware for Python 3 with the "examples" or "test" option, as recommended in the chapter on *Installation*, this should be already the case. First we'll use this library to find the sizes of the images, and later we will use it to create thumbnails.

We'll develop the application in two iterations.

7.5.1 Iteration 1: Displaying files

For simplicity, we will store image files in a subdirectory `Pics` of the default context directory `WebwareTest/Context` and let the development server deliver the files. In a production environment, you would place the `Pics` directory outside of the context and let the web server deliver the files directly.

For the first iteration, we'll display files that you upload by hand to the `Pics` directory.

We do this with two servlets – one servlet `Main.py` to show the entire album, and another `View.py` for individual pictures. Place these two servlets in the default context directory. First, `Main.py` (replacing the example servlet that has already been crated):

```python
import os

from PIL import Image  # this is from the Pillow library

from Page import Page  # the base class for web pages

dir = os.path.join(os.path.dirname(__file__), 'Pics')


class Main(Page):

    def title(self):
        # It's nice to give a real title, otherwise "Main" would be used.
```

(continues on next page)

(continued from previous page)

```
        return 'Photo Album'

    def writeContent(self):
        # We'll format these simply, one thumbnail per line:
        for filename in os.listdir(dir):
            im = Image.open(os.path.join(dir, filename))
            w, h = im.size
            # Here we figure out the scaled-down size of the image,
            # so that we preserve the aspect ratio. We'll use fake
            # thumbnails, where the image is scaled down by the browser.
            w, h = int(round(w * 100 / h)), 100
            # Note that we are just using f-strings to generate the HTML.
            # There's other ways, but this works well enough.
            # We're linking to the View servlet which we'll show later.
            # Notice we use urlencode -- otherwise we'll encounter bugs if
            # there are file names with spaces or other problematic chars.
            url = self.urlEncode(filename)
            self.writeln(f'<p><a href="View?filename={url}">'
                f'<img src="Pics/{url}" width="{w}" height="{h}"></a></p>')
```

The servlet `View.py` takes one URL parameter of `filename`. You can get the value of a URL parameter like `self.request().field('filename')` or, if you want a default value, you can use `self.request().field('filename', defaultValue)`. In the likely case you don't want to write `self.request()` before retrieving each value, do:

```
req = self.request()
self.write(req.field('username'))
```

If you need the request only once, you can write it even more compactly:

```
field = self.request().field
self.write(field('username'))
```

So here is our complete `View` servlet:

```
import os

from PIL import Image

from Page import Page

dir = os.path.join(os.path.dirname(__file__), 'Pics')


class View(Page):

    def title(self):
        return 'View: ' + self.htmlEncode(self.request().field('filename'))

    def writeContent(self):
        filename = self.request().field('filename')
        im = Image.open(os.path.join(dir, filename))
        wr = self.writeln
```

(continues on next page)

(continued from previous page)

```
        wr('<div style="text-align:center">')
        wr(f'<h4>{filename}</h4>')
        url = self.urlEncode(filename)
        w, h = im.size
        wr(f'<img src="Pics/{url}" width="{w}" height="{h}">')
        wr('<p><a href="Main">Return to Index</a></p>')
        wr('</div>')
```

7.5.2 Iteration 2: Uploading Files

That was fairly simple – but usually you want to upload files, potentially through a web interface. Along the way we'll add thumbnail generation using Pillow, and slightly improve the image index.

We'll generate thumbnails kind of on demand, so you can still upload files manually – thumbnails will be put in the directory `Thumbs` and have `-tn` appended to the name just to avoid confusion:

```
import os

from PIL import Image

from Page import Page

baseDir = os.path.dirname(__file__)
picsDir = os.path.join(baseDir, 'Pics')
thumbsDir = os.path.join(baseDir, 'Thumbs')


class Main(Page):

    def title(self):
        return 'Photo Album'

    def writeContent(self):
        # The heading:
        self.writeln(f'<h1 style="text-align:center">{self.title()}</h1>')
        # We'll format these in a table, two columns wide
        self.writeln('<table width="100%">')
        col = 0  # will be 0 for the left and 1 for the right column
        filenames = os.listdir(picsDir)
        # We'll sort the files, case-insensitive
        filenames.sort(key=lambda filename: filename.lower())
        for filename in filenames:
            if not col:  # left column
                self.write('<tr style="text-align:center">')
            thumbFilename = os.path.splitext(filename)
            thumbFilename = '{}-tn{}'.format(*thumbFilename)
            if not os.path.exists(os.path.join(thumbsDir, thumbFilename)):
                # No thumbnail exists -- we have to generate one
                if not os.path.exists(thumbsDir):
                    # Not even the Thumbs directory exists -- make it
                    os.mkdir(thumbsDir)
```

(continues on next page)

(continued from previous page)

```
                im = Image.open(os.path.join(picsDir, filename))
                im.thumbnail((250, 100))
                im.save(os.path.join(thumbsDir, thumbFilename))
            else:
                im = Image.open(os.path.join(thumbsDir, thumbFilename))
            url = self.urlEncode(filename)
            w, h = im.size
            size = os.stat(os.path.join(picsDir, filename)).st_size
            self.writeln(f'<td><p><a href="View?filename={url}">'
                f'<img src="Pics/{url}" width="{w}" height="{h}"></a></p>'
                f'<p>Filename: {filename}<br>Size: {size} Bytes</p>')
            if col:  # right column
                self.writeln('</tr>')
            col = not col
        self.write('</table>')
        self.write('<p style="text-align:center">'
            '<a href="Upload">Upload an image</a></p>')
```

In a real application, you would probably style the image more nicely using CSS, maybe using a flexbox or grid layout instead of using a table. You can add a CSS style sheet for this purpose with the `writeStyleSheet()` method.

The `View` servlet we'll leave just like it was.

We'll add an `Upload` servlet. Notice we use `enctype="multipart/form-data"` in the `<form>` tag – this is an HTMLism for file uploading (otherwise you'll just get the filename and not the file contents). Finally, when the form is finished and we have uploaded the image, we redirect them to the viewing page by using `self.response().sendRedirect(url)`:

```
import os

from Page import Page

dir = os.path.join(os.path.dirname(__file__), 'Pics')


class Upload(Page):

    def writeContent(self):
        if self.request().hasField('imageFile'):
            self.doUpload()
            return

        self.writeln('''
        <h3>Upload your image:</h3>
        <form action="Upload" method="post" enctype="multipart/form-data">
        <input type="file" name="imageFile">
        <input type="submit" value="Upload">
        </form>''')

    def doUpload(self):
        file = self.request().field('imageFile')
        # Because it's a file upload, we don't get a string back.
        # So to get the value we do this:
```

(continues on next page)

(continued from previous page)

```
        filename, contents = file.filename, file.value
        open(os.path.join(dir, filename), 'wb').write(contents)
        url = 'View?filename=' + self.urlEncode(filename)
        self.response().sendRedirect(url)
```

Using the “upload” button it should now be possible to upload images to the `Pics` directory.

CHAPTER

EIGHT

APPLICATION DEVELOPMENT

Webware provides Python classes for generating dynamic content from a web-based, server-side application. It is a significantly more powerful alternative to CGI scripts for application-oriented development.

In this chapter we explain some more fundamental concepts of Webware for Python and describe best practices for developing a web application using Webware.

8.1 Core Concepts

The core concepts of Webware for Python are the Application, Servlet, Request, Response and Transaction, for which there are one or more Python classes.

The application resides on the server-side and manages incoming requests in order to deliver them to servlets which then produce responses that get sent back to the client. A transaction is a simple container object that holds references to all of these objects and is accessible to all of them.

Content is normally served in HTML format over an HTTP connection. However, applications can provide other forms of content and the framework is designed to allow new classes for supporting protocols other than HTTP.

In order to connect the web server and the application, Webware for Python 3 uses the Web Server Gateway Interface (WSGI). The Webware Application instance serves as the WSGI callable. The web server calls the Application, passing a dictionary containing CGI-style environment variables for every request. The Application then then processes the request and sends the response back to the web server, for which WSGI provides to different mechanisms. By default, Webware applications use the `write()` callable mechanism, because this is more suitable for the way Webware for Python applications create responses, by writing to an output stream. However, since not all WSGI servers support this mechanism, it is also possible to use the more usual WSGI mechanism of passing the response as an iterable. You will need to switch the `Application.config` setting `WSGIWrite` to `False` in order to use this mechanism.

Many different WSGI servers are available that can be used with Webware for Python 3. By default, Webware uses the waitress WSGI server as its development server. If you have installed Webware with the “development” or “test” extra, as recommended in the chapter on *Installation*, the waitress server should already be installed together with Webware for Python and will be used when running the `webware serve` command.

In production, you may want to use a web server with better performance. In the chapter on *Deployment* we describe how you can configure a web server like Apache to serve static files directly, while passing dynamic contents to the Webware application via WSGI, using the mod_wsgi module.

The whole process of serving a page with Webware for Python then looks like this:

- A user requests a web page by typing a URL or submitting a form.
- The user’s browser sends the request to the remote Apache web server.
- The Apache web server passes the request to a mod_wsgi daemon process.

- The mod_wsgi daemon process collects information about the request and sends it to the Webware Application using the WSGI protocol.
- The Webware Application dispatch the raw request.
- The application instantiates an HTTPRequest object and asks the appropriate Servlet (as determined by examining the URL) to process it.
- The servlet generates content into a given HTTPResponse object, whose content is then sent back via WSGI to the mod_wsgi daemon process.
- The mod_wsgi daemon process sends the content through the web server and ultimately to the user's web browser.

8.2 Setting up your application

The first task in developing an application is to set up the file structure in which you will be working.

It is possible to put your application in a subdirectory at the path where the `webware` package is installed and change `Configs/Application.config` to add another context. But *do not do this*. Your application will be entwined with the Webware installation, making it difficult to upgrade Webware, and difficult to identify your own files from Webware files.

8.2.1 Creating a working directory

Instead you should use the `webware make` command to create an application working directory. You should run it like:

```
webware make -c Context -l Lib WorkDir
```

This will create a directory `WorkDir` that will contain a directory structure for your application. The options are:

`-c Context`:
Use `Context` as the name for the application default context. A subdirectory with the same name will be created in the work dir (you can change that with the `-d` option). If you do not use the `-c` option, the context name will be `MyContext`. You may simply want to call it `App` or `Context`, particularly if you are using only one context. If you want to add more than one context, you need to create a subdirectory and a corresponding `Contexts` dictionary entry in the `Application.config` file manually.

`-l Lib`:
Create a `Lib` directory in the work dir which will be added to the Python module search path. You can use the `-l` option multiple times; and you can also add already existent library directories outside of the work dir. If you want to add the work dir itself to the Python path, pass `-l .`. In that case, you can import from any Python package placed directly in the working, including the Webware contexts. Note that the webware package will always be added to the Python module search path, so that you can and should import any Webware modules and sub packages directly from the top level.

`WorkDir`:
The files will be put here. Name if after your application, place it where it is convenient for you. It makes sense to put the working directory together with the virtual environment where you installed Webware for Python inside the same distinct base directory. Install any other requirements either into the virtual environment or provide them in one of the directories specified with the `-l` option.

You can see all available options if you run `webware make --help`.

When you run the `webware make` command with the options describe above, the following directory structure will be created inside the `WorkDir` directory:

```
Cache/          Context/        ErrorMsgs/      Logs/           Sessions/
Configs/        error404.html   Lib/            Scripts/        Static/
```

Here's what the files and directories are for:

`Cache`:
: A directory containing cache files. You won't need to look in here.

`Configs`:
: Configuration files for the application. These files are copied from the `Configs` subdirectory in the `webware` package, but are specific to this application.

`Context`:
: The directory for your default context. This is where you put your servlets. You can change its name and location with the `-c` and `-d` options. You can also change this subsequently in the `Application.config` file in the `Configs` directory, where you can also configure more than one context. You may also want to remove the other standard contexts that come with Webware from the config file.

`error404.html`:
: The static HTML page to be displayed when a page is not found. You can remove this to display a standard error message, modify the page according to your preferences, or use a custom error servlet instead by setting `ErrorPage` in the `Application.config` file appropriately.

`ErrorMsgs`:
: HTML pages for any errors that occur. These can pile up and take up considerable size (even just during development), so you'll want to purge these every so often.

`Lib`:
: An example for an application-specific library package that can be created `-l` option (in this case, `-l Lib`).

`Logs`:
: Logs of accesses.

`Scripts`:
: This directory contains a default WSGI script named `WSGIScript.py` that can be used to start the development server or connect the Webware application with another WSGI server.

`Sessions`:
: Users sessions. These should be cleaned out automatically, you won't have to look in this directory.

`Static`:
: This directory can be used as a container for all your static files that are used by your application, but should be served directly via the web server.

8.2.2 Using a Version Control system for Your Application

A version control system is a useful tool for managing your application. Currently, Git is the most popular one. These systems handle versioning, but they also make it possible for other people to see snapshots of your progress, for multiple developers to collaborate and work on an application simultaneously, and they create a sort of implicit file share for your project. Even if you are the only developer on an application, a version control system can be very helpful.

The working directory is a good place to start for creating a versioned project. Assuming you're using Git, you can get started by creating a repository and importing your project into the repository simply by running:

```
cd WorkDir
git init
git add .
git commit -m "initial import"
```

Note that a hidden `.gitignore` file with reasonable defaults has already been created for you in the working directory. It tells Git to ignore files with certain extensions (such as `.log` or `.pyc` files), and all the files in certain directories (`Cache`, `ErrorMsgs`, `Logs`, and `Sessions`).

8.3 Structuring your Code

Once you've got the basic files and directories in place, you're ready to go in and write some code. Don't let this document get in the way of developing the application how you choose, but here are some common patterns that have proven useful for Webware applications.

8.3.1 SitePage

Subclass a `SitePage` from `Page` for your application. This subclass will change some methods and add some new methods. It serves as the basis and as a template for all the pages that follow. If you have added a `Lib` subdirectory to your working directory as explained above, place the `SitePage.py` file containing the `SitePage` class into that directory.

Some code you may wish to include in your `SitePage`:

- Authentication and security
- Accessing common objects (e.g., a user object, or a document object)
- Page header and footer
- Common layout commands, like `writeHeader`
- Database access

You may also want to add other frequently used functions into the `SitePage` module and then do `from SitePage import *` in each servlet. You can also put these functions in a dedicated `SiteFuncs` module, or distribute them in different modules, and import them explicitly, for better code readability and to avoid cluttering your namespace.

Whether you want to use functions or methods is up to you – in many cases methods can be more easily extended or customized later, but sometimes method use can become excessive and create unnecessary dependencies in your code.

A basic framework for your SitePage might be:

```
from Page import Page


class SitePage(Page):

    def respond(self, trans):
        if self.securePage():
            if not self.session().value('username', False):
                self.respondLogIn()
                return

    def securePage(self):
        """Override this method in your servlets to return True if the
        page should only be accessible to logged-in users -- by default
        pages are publicly viewable"""
        return False

```

(continues on next page)

(continued from previous page)

```
    def respondLogin(self):
        # Here we should deal with logging in...
        pass
```

Obviously there are a lot of details to add in on your own which are specific to your application and the security and user model you are using.

8.4 Configuration

There are several configuration parameters through which you can alter how Webware behaves. They are described below, including their default values. Note that you can override the defaults by placing config files in the `Configs/` directory. A config file simply contains Python code assigning the settings you wish to override. For example:

```
SessionStore = 'Memory'
ShowDebugInfoOnErrors =  True
```

See the chapter on *Configuration* for more information on settings.

8.5 Contexts

Webware divides the world into *contexts*, each of which is a directory with its own files and servlets. Webware will only serve files out of its list of known contexts.

Some of the contexts you will find out of the box are `Examples`, `Documentation` and `Admin`. When viewing either an example or admin page, you will see a sidebar that links to all the contexts.

Another way to look at contexts is a means for "directory partitioning". If you have two distinct web applications (for example, `PythonTutor` and `DayTrader`), you will likely put each of these in their own context. In this configuration, both web applications would be served by the same Application instance. Note that there may be also reasons to run multiple Application instances for serving your web applications. For instance, this would allow you to start and stop them independently, run them under different users to give them different permissions, or partition resources like number of threads individually among the web applications.

Instead of adding your own contexts you may wish to use the `webware make` command, which will partition your application from the Webware installation.

To add a new context, add to the `Contexts` dictionary of `Application.config`. The key is the name of the context as it appears in the URL and the value is the path (absolute or relative to the application working directory). Often the name of the context and the name of the directory will be the same:

```
'DayTrader': '/All/Web/Apps/DayTrader',
```

The URL to access DayTrader would then be something like: `http://localhost:8080/DayTrader/`

The special name `default` is reserved to specify what context is served when none is specified (as in `http://localhost:8080/`). Upon installation, this is the `Examples` context, which is convenient during development since it provides links to all the other contexts.

Note that a context can contain an `__init__.py` which will be executed when the context is loaded at Application startup. You can put any kind of initialization code you deem appropriate there.

8.6 Plug-ins

A plug-in is a software component that is loaded by Webware in order to provide additional functionality without necessarily having to modify Webware's source.

The most infamous plug-in is PSP (Python Server Pages) which ships with Webware.

Plug-ins often provide additional servlet factories, servlet subclasses, examples and documentation. Ultimately, it is the plug-in author's choice as to what to provide and in what manner.

Technically, plug-ins are Python packages that follow a few simple conventions in order to work with Webware. See the chapter on *Plug-ins* for information about writing your own.

8.7 Sessions

Webware provides a Session utility class for storing data on the server side that relates to an individual user's session with your site. The `SessionStore` setting determines where the data is stored and can currently be set to `Dynamic`, `File`, `Memcached`, `Memory`, `Redis` or `Shelve`.

Storing to the Dynamic session store is the fastest solution and is the default. This session storage method keeps the most recently used sessions in memory, and moves older sessions to disk periodically. All sessions will be moved to disk when the server is stopped. Note that this storage mechanism cannot be used in a multi-process environment, i.e. when you're running multiple Applications instances in different processes in production.

There are two settings in `Application.config` relating to this Dynamic session store. `MaxDynamicMemorySessions` specifies the maximum number of sessions that can be in memory at any one time. `DynamicSessionTimeout` specifies after what period of time sessions will be moved from memory to file. (Note: this setting is unrelated to the `SessionTimeout` setting below. Sessions which are moved to disk by the Dynamic Session store are not deleted). Alternatively to the Dynamic store, you can try out the Shelve session store which stores the sessions in a database file using the Python shelve module.

If you are using more than one Application instance for load-balancing, the Memcached store will be interesting for you. Using the python-memcached interface, it is able to connect to a Memcached system and store all the session data there. This allows user requests to be freely moved from one server to another while keeping their sessions, because they are all connected to the same memcache. Alternatively, using the redis-py client, the application can also store sessions in a Redis database.

All on-disk session information is located in the `Sessions` subdirectory of the application working directory.

Also, the `SessionTimeout` setting lets you set the number of minutes of inactivity before a user's session becomes invalid and is deleted. The default is 60. The Session Timeout value can also be changed dynamically on a per session basis.

8.8 Actions

Suppose you have a web page with a form and one or more buttons. Normally, when the form is submitted, a method such as Servlet's `respondToPost()` or Page's `writeBody()`, will be invoked. However, you may find it more useful to bind the button to a specific method of your servlet such as `new()`, `remove()` etc. to implement the command, and reserve `writeBody()` for displaying the page and the form that invokes these methods. Note that your "command methods" can then invoke `writeBody()` after performing their task.

The *action* feature of `Page` let's you do this. The process goes like this:

1. Add buttons to your HTML form of type `submit` and name `_action_`. For example:

```
<input name="_action_" type="submit" value="New">
<input name="_action_" type="submit" value="Delete">
```

2. Alternately, name the submit button `_action_methodName`. For example:

```
<input name="_action_New" type="submit" value="Create New Item">
```

3. Add an `actions()` method to your class to state which actions are valid. (If Webware didn't force you to do this, someone could potentially submit data that would cause any method of your servlet to be run). For example:

```
def actions(self):
    return SuperClass.actions(self) + ['New', 'Delete']
```

4. Now you implement your action methods.

The `ListBox` example shows the use of actions (in `Examples/ListBox.py`).

Note that if you proceed as in 1., you can add a `methodNameForAction()` method to your class transforming the value from the submit button (its label) to a valid method name. This will be needed, for instance, if there is a blank in the label on the button. However, usually it's simpler to proceed as in 2. in such cases.

8.9 Naming Conventions

Cookies and form values that are named with surrounding underscores (such as `_sid_` and `_action_`) are generally reserved by Webware and various plugins and extensions for their own internal purposes. If you refrain from using surrounding underscores in your own names, then (a) you won't accidentally clobber an already existing internal name and (b) when new names are introduced by future versions of Webware, they won't break your application.

8.10 Errors and Uncaught Exceptions

One of the conveniences provided by Webware is the handling of uncaught exceptions. The response to an uncaught exception is:

- Log the time, error, script name and traceback to standard output.
- Display a web page containing an apologetic message to the user.
- Save a technical web page with debugging information so that developers can look at it after-the-fact. These HTML-based error messages are stored one-per-file, if the `SaveErrorMessages` setting is true (the default). They are stored in the directory named by the `ErrorMessagesDir` (defaults to `"ErrorMsgs"`).
- Add an entry to the error log, found by default in `Logs/Errors.csv`.
- E-mail the error message if the `EmailErrors` setting is true, using the settings `ErrorEmailServer` and `ErrorEmailHeaders`. See *Configuration* for more information. You should definitely set these options when deploying a web site.

Archived error messages can be browsed through the *administration* page.

Error handling behavior can be configured as described in *Configuration*.

8.11 Activity Log

Three options let you control:

- Whether or not to log activity (`LogActivity`, defaults to 0, i.e. off)
- The name of the file to store the log (`ActivityLogFilename`, defaults to `Activity.csv`)
- The fields to store in the log (`ActivityLogColumns`) </ul>

See the chapter on *Configuration* for more information.

8.12 Administration

Webware has a built-in administration page that you can access via the `Admin` context. You can see a list of all contexts in the sidebar of any `Example` or `Admin` page.

The admin pages allows you to view Webware's configuration, logs, and servlet cache, and perform actions such as clearing the cache or reloading selected modules.

More sensitive pages that give control over the application require a user name and password, the username is `admin`, and you can set the password with the `AdminPassword` setting in the `Application.config` file.

The administration scripts provide further examples of writing pages with Webware, so you may wish to examine their source in the `Admin` context.

8.13 Debugging

8.13.1 Development Mode

When creating the Application instance, it takes a `development` flag as argument that defines whether it should run in "development mode" or "production mode". By default, if no such flag is passed, Webware checks whether the environment varibale `WEBWARE_DEVELOPMENT` is set and not empty. When you run the development server using the `webware serve` command, the flag is automatically set, so you are running in "development mode", unless you add the `--prod` option on the command line. The development flag is also available with the name `Development` in the `Application.config` file and used to make some reasonable case distinctions depending on whether the application is running in development mode. For instance, debugging information is only shown in development mode.

8.13.2 print

The most common technique is the infamous `print` statement which has been replaced with a `print()` function in Python 3. The results of `print()` calls go to the console where the WSGI server was started (not to the HTML page as would happen with CGI). If you specify `AppLogFilename` in `Application.config`, this will cause the standard output and error to be redirected to this file.

For convenient debugging, the default `Application.config` file already uses the following conditional setting:

```
AppLogFilename = None if Development else 'Application.log'
```

This will prevent standard output and error from being redirected to the log file in development mode, which makes it easier to find debugging output, and also makes it possible to use `pdb` (see below).

Prefixing the debugging output with a special tag (such as >>) is useful because it stands out on the console and you can search for the tag in source code to remove the print statements after they are no longer useful. For example:

```
print('>> fields =', self.request().fields())
```

8.13.3 Raising Exceptions

Uncaught exceptions are trapped at the application level where a useful error page is saved with information such as the traceback, environment, fields, etc. In development mode, you will see this error page directly. In production, you can examine the saved page, and you can also configure the application to automatically e-mail you this information.

When an application isn't behaving correctly, raising an exception can be useful because of the additional information that comes with it. Exceptions can be coupled with messages, thereby turning them into more powerful versions of the `print()` call. For example:

```
raise Exception(f'self = {self}')
```

While this is totally useful during development, giving away too much internal information is also a security risk, so you should make sure that the application is configured properly and no such debugging output is ever shown in production.

8.13.4 Reloading the Development Server

When a servlet's source code changes, it is reloaded. However, ancestor classes of servlets, library modules and configuration files are not. You may wish to enable the auto-reloading feature when running the development server, by adding the `-r` or `--reload` option to the `webware serve command` in order to mitigate this problem.

In any case, when having problems, consider restarting the development server (or the WSGI server you are running in production).

Another option is to use the AppControl page of the `Admin` context to clear the servlet instance and class cache (see *Administration*).

8.13.5 Assertions

Assertions are used to ensure that the internal conditions of the application are as expected. An assertion is equivalent to an `if` statement coupled with an exception. For example:

```
assert shoppingCart.total() >= 0, \
    f'shopping cart total is {shoppingCart.total()}'
```

8.13.6 Debugging using PDB

To use Python's built-in debugger `pdb`, see the tip above about setting `AppLogFilename` for convenient debugging.

To have Webware automatically put you into pdb when an exception occurs, set this in your `Application.config` file:

```
EnterDebuggerOnException = Development
```

A quick and easy way to debug a particular section of code is to add these lines at that point in the code:

```
import pdb
pdb.set_trace()
```

8.13.7 Debugging in an IDE

You can also use PyCharm or other Python IDEs to debug a Webware application. To do this, first configure the IDE to use the virtual environment where you installed Webware for Python.

Then, create the following script `serve.py` on the top level of your application working directory:

```
#!/usr/bin/python3

from webware.Scripts.WebwareCLI import main

main(['serve'])
```

Now run this file in your IDE in debug mode. For instance, in PyCharm, right-click on `serve.py` and select "Debug 'serve'".

Some IDEs like PyCharm can also debug remote processes. This could be useful to debug a test or production server.

8.14 Bootstrap Webware from Command line

You may be in a situation where you want to execute some part of your Webware applicaton from the command line, for example to implement a cron job or maintenance script. In these situations you probably don't want to instantiate a full-fledged *Application* – some of the downsides are that doing so would cause standard output and standard error to be redirected to the log file, and that it sets up the session sweeper, task manager, etc. But you may still need access to plugins such as MiscUtils, MiddleKit, which you may not be able to import directly.

Here is a lightweight approach which allows you to bootstrap Webware and plugins:

```
import webware
app = webware.mockAppWithPlugins()

# now plugins are available...
import MiscUtils
import MiddleKit
```

8.15 How do I Develop an App?

The answer to that question might not seem clear after being deluged with all the details. Here's a summary:

- Make sure you can run the development server. See the *Quickstart* for more information.
- Go through the *Beginner Tutorial*.
- Read the source to the examples (in the `Examples` subdirectory), then modify one of them to get your toes wet.
- Create your own new example from scratch. Ninety-nine percent of the time you will be subclassing the `Page` class.
- Familiarize yourself with the class docs in order to take advantage of classes like Page, HTTPRequest, HTTPResponse and Session.
- With this additional knowledge, create more sophisticated pages.
- If you need to secure your pages using a login screen, you'll want to look at the SecurePage, LoginPage, and SecureCountVisits examples in `Examples`. You'll need to modify them to suit your particular needs.

CHAPTER

NINE

CONFIGURATION

9.1 Application.config

The settings for the Application and a number of components that use it as a central point of configuration, are specified in the `Application.config` file in the `Configs` directory of the application working directory.

9.1.1 General Settings

`Contexts`:
This dictionary maps context names to the directory holding the context content. Since the default contexts all reside in Webware, the paths are simple and relative. The context name appears as the first path component of a URL, otherwise `Contexts['default']` is used when none is specified. When creating your own application, you will add a key such as `"MyApp"` with a value such as `"/home/apps/MyApp"`. That directory will then contain content such as Main.py, SomeServlet.py, SomePage.psp, etc. `webware make` will set up a context for your use as well. Default:

```
{
    'default':   'Examples',
    'Admin':     'Admin',
    'Examples':  'Examples',
    'Docs':      'Docs',
    'Testing':   'Testing',
}
```

`AdminPassword`:
The password that, combined with the `admin` id, allows access to the `AppControl` page of the `Admin` context. Set interactively when `install.py` is run (no default value).

`PrintConfigAtStartUp`:
Print the configuration to the console when the Application starts. Default: `True` (on).

`PlugIns`:
Loads the plug-ins with the given names when starting the Application. Default: `['MiscUtils', 'WebUtils', 'TaskKit', 'UserKit', 'PSP']`.

`CheckInterval`:
The number of virtual instructions after which Python will check for thread switches, signal handlers, etc. This is passed directly to `sys.setcheckinterval()` if not set to `None`. Default: `None`.

`ResponseBufferSize`:
Buffer size for the output response stream. This is only used when a servlet has set `autoFlush` to True using

the `flush()` method of the Response. Otherwise, the whole response is buffered and sent in one shot when the servlet is done. Default: `8192`.

`WSGIWrite`:
If this is set to True, then the write() callable is used instead of passing the response as an iterable, which would be the standard WSGI mechanism. Default: `True`.

`RegisterSignalHandler`:
When the Application is regularly shut down, it tries to save its Sessions and stop the TaskManager. An atexit-handler will do this automatically. You can also shut down the Application manually by calling its `shutDown()` method. If this setting is set to True, then the Application will also register signal handlers to notice when it is shutdown and shut down cleanly. However, as the `mod_wsgi` documentation explains (see section on WSGIRestrictSignal), "a well behaved Python WSGI application should not in general register any signal handlers of its own using `signal.signal()`. The reason for this is that the web server which is hosting a WSGI application will more than likely register signal handlers of its own. If a WSGI application were to override such signal handlers it could interfere with the operation of the web server, preventing actions such as server shutdown and restart." Therefore, the default setting is: `False`.

9.1.2 Path Handling

These configuration settings control which files are exposed to users, which files are hidden, and some of how those files get chosen.

`DirectoryFile`:
The list of basic filenames that Webware searches for when serving up a directory. Note that the extensions are absent since Webware will look for a file with any appropriate extension (`.py`., `.html`, `.psp`, etc). Default: `["index", "Main"]`.

`ExtensionsForPSP`:
This is the list of extensions for files to be parsed as PSP. Default: `['.psp']`.

`ExtensionsToIgnore`:
This is a list or set of extensions that Webware will ignore when autodetecting extensions. Note that this does not prevent Webware from serving such a file if the extension is given explicitly in a URL. Default: `{'.pyc', '.pyo', '.py~', '.bak'}`.

`ExtensionsToServe`:
This is a list of extensions that Webware will use exclusively when autodetecting extensions. Note that this does not prevent Webware from serving such a file if it is named explicitly in a URL. If no extensions are given all extensions will be served (usually anything but `.py` and `.psp` will be served as a static file). Default: `[]`.

`UseCascadingExtensions`:
If False, Webware will give a `404 Not Found` result if there is more than one file that could potentially match. If True, then Webware will use the `ExtensionCascadeOrder` setting to determine which option to serve. Default: `True`.

`ExtensionCascadeOrder`:
A list of extensions that Webware will choose, in order, when files of the same basename but different extensions are available. Note that this will have no effect if the extension is given in the URL. Default: `[".psp", ".py", ".html"]`.

`FilesToHide`:
A list or set of file patterns to protect from browsing. This affects all requests, and these files cannot be retrieved even when the extension is given explicitly. Default: `{".*", "*~", "*bak", "*.tmpl", "*.pyc", "*.pyo", "*.config"}`.

`FilesToServe`:
File patterns to serve from exclusively. If the file being served for a particular request does not match one of

these patterns an `HTTP 403 Forbidden` error will be return. This affects all requests, not just requests with auto detected extensions. If set to `[]` then no restrictions are placed. Default: `[]`.

9.1.3 Sessions

`MemcachedNamespace`:
The namespace used to prefix all keys from the Webware application when accessing Memcached servers for storing sessions. You should change this if you are using the same memcache for different applications. Default: `'WebwareSession:'`.

`MemcachedOnIteration`:
This setting determines how Webware behaves when attempting to iterate over the sessions or clear the session store, when using `Memcached`. If you set it to `Error`, this will raise an Exception, when set to `Warning`, it will print a Warning, when set to `None`, it will be ignored (the size of the session store will be always reported as zero). Default: `Warning`.

`MemcachedServers`:
This sets the list of Memcached servers used when setting `SessionStore` to `Memcached`. Default: `['localhost:11211']`.

`RedisNamespace`:
The namespace used to prefix all keys from the Webware application when accessing Redis servers for storing sessions. You should change this if you are using the same Redis instance for different applications. Default: `'WebwareSession:'`.

`RedisHost`:
This sets the Redis host that shall be used when setting `SessionStore` to `Redis`. Default: `'localhost'`.

`RedisPort`:
This sets the port for the Redis connection that shall be used when setting `SessionStore` to `Redis`. Default: `6379`.

`RedisDb`:
This sets the database number for the Redis connection that shall be used when setting `SessionStore` to `Redis`. Default: `0`.

`RedisPassword`:
This sets the password for the Redis connection that shall be used when setting `SessionStore` to `Redis`. Default: `None`.

`SessionModule`:
Can be used to replace the standard Webware Session module with something else. Default: `Session`

`SessionStore`:
This setting determines which of five possible session stores is used by the Application: `Dynamic`, `File`, `Memcached`, `Memory`, `Redis` or `Shelve`. The `File` store always gets sessions from disk and puts them back when finished. `Memory` always keeps all sessions in memory, but will periodically back them up to disk. `Dynamic` is a good cross between the two, which pushes excessive or inactive sessions out to disk. `Shelve` stores the sessions in a database file using the Python `shelve` module, `Memcached` stores them on a Memcached system using the `python-memcached` interface, and `Redis` stores them on a Redis system using the `redis-py` client. You can use a custom session store module as well. Default: `Dynamic`.

`SessionStoreDir`:
If `SessionStore` is set to `File`, `Dynamic` or `Shelve`, then this setting determines the directory where the files for the individual sessions or the shelve database will be stored. The path is interpreted as relative to the working directory (or Webware path, if you're not using a working directory), or you can specify an absolute path. Default: `Sessions`.

`SessionTimeout`:
Determines the amount of time (expressed in minutes) that passes before a user's session will timeout. When a session times out, all data associated with that session is lost. Default: `60`.

`AlwaysSaveSessions`:
If False, then sessions will only be saved if they have been changed. This is more efficient and avoids problems with concurrent requests made by the same user if sessions are not shared between these requests, as is the case for session stores other than `Memory` or `Dynamic`. Note that in this case the last access time is not saved either, so sessions may time out if they are not altered. You can call `setDirty()` on sessions to force saving unaltered sessions in this case. If True, then sessions will always be saved. Default: `True`.

`IgnoreInvalidSession`:
If False, then an error message will be returned to the user if the user's session has timed out or doesn't exist. If True, then servlets will be processed with no session data. Default: `True`.

`UseAutomaticPathSessions`:
If True, then the Application will include the session ID in the URL by inserting a component of the form `_SID_=8098302983` into the URL, and will parse the URL to determine the session ID. This is useful for situations where you want to use sessions, but it has to work even if the users can't use cookies. If you use relative paths in your URLs, then you can ignore the presence of these sessions variables. The name of the field can be configured with the setting `SessionName`. Default: `False`.

`UseCookieSessions`:
If True, then the application will store the session ID in a cookie with the name set in `SessionName`, which is usually `_SID_`. Default: `True`.

`SessionCookiePath`:
You can specify a path for the session cookie here. `None` means that the servlet path will be used, which is normally the best choice. If you rewrite the URL using different prefixes, you may have to specify a fixed prefix for all your URLs. Using the root path '/' will always work, but may have security issues if you are running less secure applications on the same server. Default: `None`.

`SecureSessionCookie`:
If True, then the Application will use a secure cookie for the session ID if the request was using an HTTPS connection. Default: `True`.

`HttpOnlySessionCookie`:
If True, then the Application will set the HttpOnly attribute on the session cookie . Default: `True`.

`SameSiteSessionCookie`:
If not `None`, then the Application will set this value as the SameSite attribute on the session cookie . Default: `Strict`.

`MaxDynamicMemorySessions`:
The maximum number of dynamic memory sessions that will be retained in memory. When this number is exceeded, the least recently used, excess sessions will be pushed out to disk. This setting can be used to help control memory requirements, especially for busy sites. This is used only if the `SessionStore` is set to `Dynamic`. Default: `10000`.

`DynamicSessionTimeout`:
The number of minutes of inactivity after which a session is pushed out to disk. This setting can be used to help control memory requirements, especially for busy sites. This is used only if the `SessionStore` is set to `Dynamic`. Default: `15`.

`SessionPrefix`:
This setting can be used to prefix the session IDs with a string. Possible values are `None` (don't use a prefix), `"hostname"` (use the hostname as the prefix), or any other string (use that string as the prefix). You can use this for load balancing, where each Webware server uses a different prefix. You can then use mod_rewrite or other

software for load-balancing to redirect each user back to the server they first accessed. This way the backend servers do not have to share session data. Default: `None`.

`SessionName`:
This setting can be used to change the name of the field holding the session ID. When the session ID is stored in a cookie and there are applications running on different ports on the same host, you should choose different names for the session IDs, since the web browsers usually do not distinguish the ports when storing cookies (the port cookie-attribute introduced with RFC 2965 is not used). Default: `_SID_`.

`ExtraPathInfo`:
When enabled, this setting allows a servlet to be followed by additional path components which are accessible via HTTPRequest's `extraURLPath()`. For subclasses of `Page`, this would be `self.request().extraURLPath()`. Default: `False`.

`UnknownFileTypes`:
This setting controls the manner in which Webware serves "unknown extensions" such as .html, .css, .js, .gif, .jpeg etc. The default setting specifies that the servlet matching the file is cached in memory. You may also specify that the contents of the files shall be cached in memory if they are not too large.

If you are concerned about performance, use mod_rewrite to avoid accessing Webware for static content.

The `Technique` setting can be switched to `"redirectSansAdapter"`, but this is an experimental setting with some known problems.

Default:

```
{
    'ReuseServlets': True,  # cache servlets in memory
    'Technique': 'serveContent',  # or 'redirectSansAdapter'
    # If serving content:
    'CacheContent': False,  # set to True for caching file content
    'MaxCacheContentSize': 128*1024,  # cache files up to this size
    'ReadBufferSize': 32*1024  # read buffer size when serving files
}
```

9.1.4 Caching

`CacheServletClasses`:
When set to False, the Application will not cache the classes that are loaded for servlets. This is for development and debugging. You usually do not need this, as servlet modules are reloaded if the file is changed. Default: `True` (caching on).

`CacheServletInstances`:
When set to False, the Application will not cache the instances that are created for servlets. This is for development and debugging. You usually do not need this, as servlet modules are reloaded and cached instances purged when the servlet file changes. Default: `True` (caching on).

`CacheDir`:
This is the name of the directory where things like compiled PSP templates are cached. Webware creates a subdirectory for every plug-in in this directory. The path is interpreted as relative to the working directory (or Webware path, if you're not using a working directory), or you can specify an absolute path. Default: `Cache`.

`ClearPSPCacheOnStart`:
When set to False, the Application will allow PSP instances to persist from one application run to the next. If you have PSPs that take a long time to compile, this can give a speedup. Default: `False` (cache will persist).

`ReloadServletClasses`:
: During development of an application, servlet classes will be changed very frequently. The AutoReload mechanism could be used to detect such changes and to reload modules with changed servlet classes, but it would cause an application restart every time a servlet class is changed. So by default, modules with servlet classes are reloaded without restarting the server. This can potentially cause problems when other modules are dependent on the reloaded module because the dependent modules will not be reloaded. To allow reloading only using the AutoReload mechanism, you can set `ReloadServletClasses` to `False` in such cases. Default: `True` (quick and dirty reloading).

9.1.5 Errors

`ShowDebugInfoOnErrors`:
: If True, then uncaught exceptions will not only display a message for the user, but debugging information for the developer as well. This includes the traceback, HTTP headers, form fields, environment and process ids. You will most likely want to turn this off when deploying the site for users. Default: `True`.

`EnterDebuggerOnException`:
: If True, and if the AppServer is running from an interactive terminal, an uncaught exception will cause the application to enter the debugger, allowing the developer to call functions, investigate variables, etc. See the Python debugger (pdb) docs for more information. You will certainly want to turn this off when deploying the site. Default: `False` (off).

`IncludeEditLink`:
: If True, an "[edit]" link will be put next to each line in tracebacks. That link will point to a file of type `application/x-webware-edit-file`, which you should configure your browser to run with `bin/editfile.py`. If you set your favorite Python editor in `editfile.py` (e.g. `editor = 'Vim'`), then it will automatically open the respective Python module with that editor and put the cursor on the erroneous line. Default: `True`.

`IncludeFancyTraceback`:
: If True, then display a fancy, detailed traceback at the end of the error page. It will include the values of local variables in the traceback. This makes use of a modified version of `cgitb.py` which is included with Webware as `CGITraceback.py`. The original version was written by Ka-Ping Yee. Default: `False` (off).

`FancyTracebackContext`:
: The number of lines of source code context to show if IncludeFancyTraceback is turned on. Default: `5`.

`UserErrorMessage`:
: This is the error message that is displayed to the user when an uncaught exception escapes a servlet. Default: `"The site is having technical difficulties with this page. An error has been logged, and the problem will be fixed as soon as possible. Sorry!"`

`ErrorLogFilename`:
: The name of the file where exceptions are logged. Each entry contains the date and time, filename, pathname, exception name and data, and the HTML error message filename (assuming there is one). Default: `Errors.csv`.

`SaveErrorMessages`:
: If True, then errors (e.g., uncaught exceptions) will produce an HTML file with both the user message and debugging information. Developers/administrators can view these files after the fact, to see the details of what went wrong. These error messages can take a surprising amount of space. Default: `True` (do save).

`ErrorMessagesDir`:
: This is the name of the directory where HTML error messages get stored. The path is interpreted as relative to the working directory, or you can specify an absolute path.Default: `ErrorMsgs`.

`EmailErrors`:
: If True, error messages are e-mailed out according to the ErrorEmailServer and ErrorEmailHeaders settings. You must also set `ErrorEmailServer` and `ErrorEmailHeaders`. Default: `False` (false/do not email).

EmailErrorReportAsAttachment:
Set to True to make HTML error reports be emailed as text with an HTML attachment, or False to make the html the body of the message. Default: `False` (HTML in body).

ErrorEmailServer:
The SMTP server to use for sending e-mail error messages, and, if required, the port, username and password, all separated by colons. For authentication via "SMTP after POP", you can furthermore append the name of a POP3 server, the port to be used and an SSL flag. Default: `'localhost'`.

ErrorEmailHeaders:
The e-mail headers used for e-mailing error messages. Be sure to configure `"From"`, `"To"` and `"Reply-To"` before turning `EmailErrors` on. Default:

```
{
    'From':         'webware@mydomain,
    'To':           ['webware@mydomain'],
    'Reply-To':     'webware@mydomain',
    'Content-Type': 'text/html',
    'Subject':      'Error'
}
```

ErrorPage:
You can use this to set up custom error pages for HTTP errors and any other exceptions raised in Webware servlets. Set it to the URL of a custom error page (any Webware servlet) to catch all kinds of exceptions. If you want to catch only particular errors, you can set it to a dictionary mapping the names of the corresponding exception classes to the URL to which these exceptions should be redirected. For instance:

```
{
     'HTTPNotFound': '/Errors/NotFound',
     'CustomError':  '/Errors/Custom'
}
```

If you want to catch any exceptions except HTTP errors, you can set it to:

```
{
    'Exception':     '/ErrorPage',
    'HTTPException': None
}
```

Whenever one of the configured exceptions is thrown in a servlet, you will be automatically forwarded to the corresponding error page servlet. More specifically defined exceptions overrule the more generally defined. You can even forward from one error page to another error page unless you are not creating loops. In an `HTTPNotFound` error page, the servlet needs to determine the erroneous URI with `self.request().previousURI()`, since the `uri()` method returns the URI of the current servlet, which is the error page itself. When a custom error page is displayed, the standard error handler will not be called. So if you want to generate an error email or saved error report, you must do so explicitly in your error page servlet. Default: `None` (no custom error page).

MaxValueLengthInExceptionReport:
Values in exception reports are truncated to this length, to avoid excessively long exception reports. Set this to `None` if you don't want any truncation. Default: `500`.

RPCExceptionReturn:
Determines how much detail an RPC servlet will return when an exception occurs on the server side. Can take the values, in order of increasing detail, `"occurred"`, `"exception"` and `"traceback"`. The first reports the string `"unhandled exception`", the second prints the actual exception, and the third prints both the exception and accompanying traceback. All returns are always strings. Default: `"traceback"`.

`ReportRPCExceptionsInWebware`:
: True means report exceptions in RPC servlets in the same way as exceptions in other servlets, i.e. in the logfiles, the error log, and/or by email. False means don't report the exceptions on the server side at all; this is useful if your RPC servlets are raising exceptions by design and you don't want to be notified. Default: `True` (do report exceptions).

9.1.6 Logging

`LogActivity`:
: If True, then the execution of each servlet is logged with useful information such as time, duration and whether or not an error occurred. Default: `True`.

`ActivityLogFilenames`:
: This is the name of the file that servlet executions are logged to. This setting has no effect if `LogActivity` is False. The path can be relative to the Webware location, or an absolute path. Default: `'Activity.csv'`.

`ActivityLogColumns`:
: Specifies the columns that will be stored in the activity log. Each column can refer to an object from the set [application, transaction, request, response, servlet, session] and then refer to its attributes using "dot notation". The attributes can be methods or instance attributes and can be qualified arbitrarily deep. Default: `['request.remoteAddress', 'request.method', 'request.uri', 'response.size', 'servlet.name', 'request.timeStamp', 'transaction.duration', 'transaction.errorOccurred']`.

`AppLogFilename`:
: The Application redirects standard output and error to this file, if this is set in production mode. Default: `'Application.log'`.

`` `LogDir ``:
: The directory where log files should be stored. All log files without an explicit path will be put here. Default: `'Logs'`.

`Verbose`:
: If True, then additional messages are printed while the Application runs, most notably information about each request such as size and response time. Default: `True`.

`SilentURIs`:
: If `Verbose` is set to True, then you can use this setting to specify URIs for which you don't want to print any messages in the output of the Application. The value is expected to be a regular expression that is compared to the request URI. For instance, if you want to suppress output for images, JavaScript and CSS files, you can set `SilentURIs` to `'\.(gif|jpg|jpeg|png|js|css)$'` (though we do not recommend serving static files with Webware; it's much more efficient to deliver them directly from the Apache server). If set to `None`, messages will be printed for all requests handled by Webware. Default: `None`

CHAPTER TEN

DEPLOYMENT

Webware for Python 3 uses the Web Server Gateway Interface (WSGI) which allows deploying Webware apps with any available WSGI server.

If your performance requirements are not that high, you can use waitress as WSGI server, which is used as the development server for Webware, to serve your application on a production system as well. If your performance requirements are higher, we recommend serving Webware applications using Apache and mod_wsgi. But there are also many other options, and you can add caching, load balancing and other techniques or use a CDN to improve performance.

10.1 Installation on the Production System

In order to install your Webware for Python 3 application on the production system, first make sure the minimum required Python 3.6 version is already installed. One popular and recommended option is running a Linux distribution on your production system - see Installing Python 3 on Linux.

Next, we recommend creating a virtual environment for your Webware for Python 3 application. We also recommend creating a dedicated user as owner of your application, and placing the virtual environment into the home directory of that user. When you are logged in as that user under Linux, you can create the virtual environment with the following command. If you get an error, you may need to install `python3-venv` as an additional Linux package before you can run this command:

```
python3 -m venv .venv
```

This will create the virtual environment in the subdirectory `.venv`. Of course, you can also use a different name for that directory. Now, install Webware for Python 3 into that virtual environment. Under Linux, you can do this as follows:

```
. .venv/bin/activate
pip install "Webware-for-Python>=3"
```

You will also need to install other Python packages required by your application into the virtual environment with pip, unless these are provided as part of the application, e.g. in a `Lib` subdirectory of the application working directory. If you want to use a Python-based WSGI server such as waitress, you need to install it into this virtual environment as well:

```
pip install waitress
```

As the next step, you need to copy the application working directory containing your Webware for Python 3 application to your production system. We recommend putting it into the directory where you created the virtual environment, so that both are siblings. It is important that the application working directory is readable for the user that will run the WSGI server, but not writable. For security reasons, we recommend running the WSGI server with as a dedicated user with low privileges who is not the owner of the application working directory. The directory should also not be readable to other users. The subdirectories of the application working directory should be readable only as well, except for the Cache, ErrorMsgs, Logs and Sessions subdirectories, which must be writable. You can use the webware make command to change the ownership of the application working directory to a certain user or group. You can also run this command on an existing working directory that you copied to the production server. For instance, assuming you activated the virtual environment with Webware for Python, and you have superuser privileges, you could make the application accessible to the group www-data like this:

```
webware make -g www-data path-to-app-work-dir
```

We recommend using an automatic deployment solution such as Fabric for copying your application working directory from your development or staging server to your production server. It is also possible to use Git hooks to deploy your application with Git.

Also, make sure the virtual environment you created above is readable by the user running the WSGI server, e.g. by using the same group ownership as above:

```
chgrp -R www-data .venv
```

10.2 Starting the WSGI Server on Boot

On a production system, you want to set up your system so that the WSGI server starts automatically when the system boots. If you are using Apache and mod_wsgi, as explained further below, then you only need to make sure Apache starts automatically, and you can skip this step.

There are a lot of options to start applications at boot time. First, you can use the startup system of your operating system directly. We will show how this works using systemd as an example. Second, you can use one of the many available process managers to start and control the WSGI server. We will show how this works using Supervisor.

10.2.1 Using systemd

We assume that you have already copied your application working directory to the production system as explained above, and we assume you're using waitress as your WSGI server. In order to make your application available as a systemd service, you only need to add the following service file into the directory /etc/systemd/system. The service file should be named something like webware.service or name-of-your-app.service if you're running multiple Webware applications:

```
[Unit]
Description=My Webware application
After=network.target
StartLimitIntervalSec=0

[Service]
Type=simple
Restart=on-failure
RestartSec=1
User=www-data
Group=www-data
ExecStart=path-to-virtual-env/bin/webware serve --prod
WorkingDirectory=path-to-app-work-dir

[Install]
WantedBy=multi-user.target
```

Adapt the options as needed. `Description` should be a meaningful description of your Webware application. With `User` and `Group` you specify under which user and group your Webware application shall run, see the remarks above. Adapt the `EexecStart` option so that it uses the path to your virtual environment, and specify the path to your application working directory as the `WorkingDirectory` option. You can change the host address, port and add other options to `webware serve` in the `ExecStart` option. By default, the server runs on port 8080, but you can specify a different port using the `-p` option. If you want to run waitress behind a reverse proxy, for instance because you want to run on port 80 which needs superuser privileges or you need TLS support which is not provided by waitress, then you you need to serve only on the local interface, using options such as `-l 127.0.0.1 -p 8080`. The `--prod` option tells Webware to run in production mode.

Note that if you use the `--reload` option with `webware serve` in `ExecStart`, then you should also set `KillMode=process` and `ExecStopPost=/bin/sleep 1` in the service file to make sure that Webware can be shut down properly.

After adding or changing the service file, you need to run the following command so that systemd refreshes its configuration:

```
sudo systemctl daemon-reload
```

You tell systemd to automatically run your service file on system boot by enabling the service with the following command:

```
sudo systemctl enable webware
```

If you named your service file differently, you need to specify that name instead of `webware` in this command. Likewise, you can disable the service with:

```
sudo systemctl disable webware
```

To start the service manually, run this command:

```
sudo systemctl start webware
```

You can list errors that appeared while running the service using this command:

```
sudo journalctl -ru webware
```

The output of your application will be logged to the file `Logs/Application.log` inside the application working directory if you did not specify anything else in the Webware application configuration.

To restart the service, you need to do this:

```
sudo systemctl restart webware
```

If you want to automatically restart the service whenever there are changes in the application working directory, you can install a systemd path unit to watch the directory and run the above command whenever something changes. Alternatively, you can run `webware serve` with the `--reload` option. In that case, you also need to install hupper into the virtual environment where you installed Webware, because it is used to implement the `reload` functionality. If you are using a deployment tool such as Fabric, you can simply run the above command after deploying the application instead of watching the directory for changes.

10.2.2 Using Supervisor

You can also use Supervisor to control your WSGI server. On many Linux distributions, Supervisor can be installed with the package manager, but you can also install it manually using:

```
pip install supervisor
```

The disadvantage of such a manual installation is that you will also need to integrate it into the service management infrastructure of your system manually, e.g. using a service file as explained above. Therefore we recommend that you install the Linux package if it is available. For instance, on Ubuntu you would do this with:

```
sudo apt-get install supervisor
```

In the following, we assume that you installed Supervisor like this. You will then usually have a directory `/etc/supervisor` with a subdirectory `conf.d`. Inside this subdirectory, create the following configuration file. The configuration file should be name something like `webware.conf` or `name-of-your-app.conf` if you're running multiple Webware applications:

```
[program:webware]
user=www-data
command=path-to-virtual-env/bin/webware serve --prod
directory=path-to-app-work-dir
```

You can add many more options to the configuration. Adapt the options above and add other options as needed. You may want to change the section header `[program:webware]` to a more specific name if you are running multiple Webware applications. The `user` options specifies which user shall run your Webware application. Adapt the `command` option so that it uses the path to your virtual environment, and specify the path to your application working directory as the `directory` option. You can change the host address, port and add other options to `webware serve` in the `command` option. By default, the server runs on port 8080, but you can specify a different port using the `-p` option. If you want to run waitress behind a reverse proxy, for instance because you want to run on port 80 which needs superuser privileges or you need TLS support which is not provided by waitress, then you you need to serve only on the local interface, using options such as `-l 127.0.0.1 -p 8080`. The `--prod` option tells Webware to run in production mode.

Reload the Supervisor configuration file and restart affected programs like this:

```
supervisorctl reread
supervisorctl update
```

This should automatically start the Webware application.

By default, the output of your application will be redirected to the file `Logs/Application.log` inside the application working directory by Webware. You can change the location of this file using the Webware application configuration, or you can also use Supervisor options to redirect the output to a log file and control that log file.

To show the process status of your application, run this command:

```
supervisorctl status webware
```

If you named the configuration section differently, you need to specify that name instead of `webware` in this command. In order to restart the application, run this command:

```
supervisorctl restart webware
```

If you want to automatically restart whenever there are changes in the application working directory, you can for example use Supervisor to run a separate program that watches the directory using inotify, and runs the above command whenever something changes, or you can run `webware serve` with the `--reload` option. In that case, as explained above, you also need to install hupper into the virtual environment where you installed Webware. If you are using a deployment tool such as Fabric, you can simply run the above command after deploying the application instead of watching the directory for changes.

10.3 Logfile Rotation

The application log file (which you will find in `Logs/Application.log` inside the application working directory by default) will increase in size over time. We recommend configuring logrotate to rotate this log file, since this does not happen automatically. On most Linux distributions, logrotate is already pre-installed and you just need to put a configuration file like this into the folder `/etc/logrotate.d`:

```
path-to-app-work-dir/Logs/Application.log {
  weekly
  rotate 9
  copytruncate
  compress
  dateext
  missingok
  notifempty
  su www-data www-data
}
```

Modify the configuration as you see fit. The `su` directive should specify the user and the group under which the WSGI server is running. Note that you can specify more than one log path and/or use wildcards, so that you can apply the same configuration to several Webware applications and avoid repeating the same options.

Assuming you created the configuration file as `/etc/logrotate.d/webware`, you can test it with this command:

```
logrotate -f /etc/logrotate.d/webware
```

10.4 Running behind a Reverse Proxy

There are several reasons why you may want to run the WSGI server that is serving your Webware application behind a reverse proxy. First, it can serve as a kind of load balancer, redirecting traffic to other applications or static files away from your Webware application and request the WSGI server only for the dynamic content where it is really needed. Second, it can provide TLS encryption in order to support HTTPS connections, compress data going in and out the server, and cache frequently used content, and is optimized to do all of this very quickly. If you're using the waitress WSGI server, this is an important issue, since waitress itself does not provide TLS support. Third, a reverse proxy also adds another security layer to your production system. In the following we show how you can use Apache and NGINX as reverse proxy for your Webware application.

Again, if you are using Apache and mod_wsgi, as explained further below, then you normally don't need a separate proxy server, and you can skip this step.

10.4.1 Using Apache as Reverse Proxy

The first thing you need to do after installing Apache is to enable the Apache mod_proxy and mod_proxy_http modules. You can usually do this as follows:

```
sudo a2enmod proxy proxy_http
```

At this point, you may want to enable other Apache modules as well. For instance, if you want to use encryption with TLS (HTTPS connections), you need to also enable the mod_ssl module:

```
sudo a2enmod ssl
```

Maybe you want to enable some more modules providing load balancing capabilities, such as mod_proxy_balancer and mod_lbmethod_byrequests. We won't cover these modules in this deployment guide, but keep in mind that they are available if you need to scale up.

Assuming you configured the WSGI server to run on port 8080 using the localhost interface 127.0.0.1, you now need to add the following directives to your Apache configuration:

```
ProxyPass / http://127.0.0.1:8080/
ProxyPassReverse / http://127.0.0.1:8080/
```

Note: Do *not* set `SSLProxyEngine On`, even if you want to communicate via HTTPS with your clients. You would only need this when the communication between Apache and the WSGI server is encrypted as well, which is usually not necessary, particularly if you run the reverse proxy and the WSGI server on the same machine, and would only work with WSGI servers that support encryption.

If you want to support encryption, you also need to create a server certificate and specify it in your Apache configuration. For testing only, a self-signed certificate will do, which may be already installed and configured. In the Internet you will find many instructions for creating a real server certificate and configuring Apache to use it.

Reload Apache after any changes you make to the configuration, e.g. with `systemctl reload apache2` or `apachectl -k graceful`.

The two lines of configuration above make Apache work as a reverse proxy for any URL, i.e. all traffic is passed on to the WSGI server. This means that the WSGI server will also deliver any static assets that are part of your application, like images, CSS scripts, JavaScript files or static HTML pages. This is inefficient and creates an unnecessary load on the WSGI server. It is much more efficient if you let Apache serve the static assets. To achieve this, use the following Apache configuration:

```
Alias /static path-to-app-work-dir/Static
<Directory path-to-app-work-dir/Static>
    Require all granted
</Directory>
ProxyPass /static !
ProxyPass / http://127.0.0.1:8080/
ProxyPassReverse / http://127.0.0.1:8080/
```

With this configuration, you can access the static assets in the `Static` subdirectory of the application working directory with the URL prefix `/static`, while everything else will be passed on to the WSGI server and handled by Webware for Python.

You can also do it the other way around, e.g. let everything behind the prefix `/app` be handled by Webware for Python, and everything else looked up as a static asset in the `Static` subdirectory of the application working directory, using a configuration like this:

```
DocumentRoot path-to-app-work-dir/Static
<Directory path-to-app-work-dir/Static>
    Require all granted
</Directory>
ProxyPass /app http://127.0.0.1:8080/
ProxyPassReverse /app http://127.0.0.1:8080/
```

In this case, you should also tell the Webware application that you are using the `/app` prefix. If you are running the waitress server with `webware serve` you can do so using the `--url-prefix` command line option:

```
webware serve -l 127.0.0.1 -p 8080 --url-prefix /app --prod
```

This prefix will then be passed to Webware in the `SCRIPT_NAME` environment variable, which is used when determining the `servletPath()` of a Webware `HTTPRequest`.

Similarly, to tell Webware that you are using HTTPS connections, you can use the `--url-scheme` command line option:

```
webware serve -l 127.0.0.1 -p 8080 --url-schema https --prod
```

You should then also add the following line to the Apache configuration:

```
RequestHeader set X-Forwarded-Proto https
```

If you want to override WSGI environment variables using proxy headers, you need to add the options `--trusted-proxy` and `trusted-proxy-headers` to the `webware serve` command.

See also the remarks on running behind a reverse proxy in the waitress documentation.

10.4.2 Using NGINX as a Reverse Proxy

Frequently, NGINX is used instead of Apache as a reverse proxy, because it is more lightweight and performs a bit better when serving static content. Contrary to Apache, you don't need to enable any additional modules to make NGINX work as a reverse proxy.

After installing NGINX and configuring the WSGI server to run on port 8080 using the localhost interface 127.0.0.1, you now need to add the following lines to your NGINX configuration:

```
location /static {
   alias path-to-app-work-dir/Static;
}

location / {
   proxy_pass http://127.0.0.1:8080/;

   proxy_set_header Host $host;
   proxy_set_header X-Forwarded-Proto $scheme;
   proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
   proxy_set_header X-Forwarded-Host $host;
   proxy_set_header X-Forwarded-Port $server_port;
   proxy_set_header X-Real-IP $remote_addr;
}
```

If you want to support encryption, you also need to create a server certificate and specify it in your NGINX configuration. For testing only, a self-signed certificate will do, which may be already installed. In the Internet you will find many instructions for creating a real server certificate and configuring NGINX to use it.

After reloading the NGINX configuration, e.g. with `nginx -s reload`, NGINX should now act as a reverse proxy and deliver your Webware application at the root URL, and static content in the `Static` subdirectory of the application working directory with the URL prefix `/static`.

If you want to do it the other way around, i.e. serve any static assets at the root URL, and your Webware application with the URL prefix `/app`, use this configuration instead:

```
root path-to-app-work-dir/Static

location / {
}

location /app {
   proxy_pass http://127.0.0.1:8080/;

   proxy_set_header Host $host;
   proxy_set_header X-Forwarded-Proto $scheme;
   proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
   proxy_set_header X-Forwarded-Host $host;
   proxy_set_header X-Forwarded-Port $server_port;
   proxy_set_header X-Real-IP $remote_addr;
}
```

In this case, you should also tell the Webware application that you are using the `/app` prefix. If you are running the waitress server with `webware serve` you can do so using the `--url-prefix` command line option:

```
webware serve -l 127.0.0.1 -p 8080 --url-prefix /app --prod
```

This prefix will then be passed to Webware in the `SCRIPT_NAME` environment variable, which is used when determining the `servletPath()` of a Webware `HTTPRequest`.

If you want to override WSGI environment variables using proxy headers, you need to add the options `--trusted-proxy` and `trusted-proxy-headers` to the `webware serve` command.

See also the remarks on running behind a reverse proxy in the waitress documentation.

10.5 Using Apache and mod_wsgi

While you can deploy Webware applications using the waitress WSGI server, as explained above, or run the application with other possibly better performing WSGI servers, as explained further below, our recommended way of deploying Webware application is using Apache and mod_wsgi, since it combines excellent performance with low installation and maintenance effort. In particular, you will not need to care about running a separate WSGI server and starting it automatically, because this is handled by mod_wsgi already, and you will not need to install a reverse proxy, because you can use Apache to server the static content and dispatch to Webware via mod_wsgi for the dynamic content. The Apache web server can also care about everything that is needed to serve your content securely via HTTPS.

The first thing you need is to make sure that Apache is installed on your production system with the “worker” MPM module. On some systems, the “prefork” MPM module is still the default, but “worker” is much better suited for our purposes. See also the section on processes and threading in the mod_wsgi documentation.

Next you will need to install mod_wsgi. If possible, install a version that is available as a binary package for your system. There may be different versions of mod_wsgi available. Make sure you install the one for the Apache version running on your system and the Python version you are using in your Webware application. The package may be called something like “apache2-mod_wsgi-python3” or “libapache2-mod-wsgi-py3”. If no suitable, current version of mod_wsgi is available, you will need to install mood_wsgi from source.

After installation, the module should be already enabled, but to be sure, enable the mod_wsgi Apache module with the following command:

```
sudo a2enmod wsgi
```

At this point, you may want to enable other Apache modules as well. For instance, if you want to use encryption with TLS (HTTPS connections), you need to also enable the mod_ssl module:

```
sudo a2enmod ssl
```

In that case, you also need to create a server certificate and specify it in your Apache configuration. For testing only, a self-signed certificate will do, which may be already installed and configured. In the Internet you will find many instructions for creating a real server certificate and configuring Apache to use it.

Add the following lines to your Apache configuration in order to serve your Webware application under the root URL, and static assets under the URL prefix `/static`:

```
Alias /static path-to-app-work-dir/Static

<Directory path-to-app-work-dir/Static>
    Require all granted
</Directory>

WSGIDaemonProcess webware threads=20 python-home=path-to-virtual-env
WSGIProcessGroup webware

WSGIScriptAlias / path-to-app-work-dir/Scripts/WSGIScript.py

<Directory path-to-app-work-dir/Scripts>
    Require all granted
</Directory>
```

Note that `path-to-virtual-env` should really be the path of the directory containing the virtual environment where you installed Webware for Python 3 and other requirements for your Webware application, not the path to the Python interpreter.

Reload Apache after any changes you make to the configuration, e.g. with `systemctl reload apache2` or `apachectl -k graceful`.

If you want to do it the other way around, i.e. serve any static assets at the root URL, and your Webware application with the URL prefix `/app`, use this configuration instead:

```
DocumentRoot path-to-app-work-dir/Static

<Directory path-to-app-work-dir/Static>
    Require all granted
</Directory>

WSGIDaemonProcess webware threads=20 python-home=path-to-virtual-env
WSGIProcessGroup webware

WSGIScriptAlias /app path-to-app-work-dir/Scripts/WSGIScript.py
```

(continues on next page)

(continued from previous page)

```

<Directory path-to-app-work-dir/Scripts>
    Require all granted
</Directory>
```

In this case, the prefix `/app` will be also passed to Webware by mod_wsgi in the `SCRIPT_NAME` environment variable, and is considered when determining the `servletPath()` of a Webware `HTTPRequest`.

You can test the Apache configuration for errors with the command `apache2ctl configtest`. To debug problems with mod_wsgi, you can also use these settings in the Apache configuration:

```
LogLevel info
WSGIVerboseDebugging On
```

A frequent problem is that the virtual environment into which you installed Webware uses a different Python version than the one that the currently enabled mod_wsgi module was built for. In this case, re-create the virtual environment with the proper Python version, or install a mod_wsgi module that was built for the Python version you are using in your Webware application.

The output of your application will be logged to the file `Logs/Application.log` inside the application working directory if you did not specify anything else in the Webware application configuration (see also *Logfile Rotation*).

Note that mod_wsgi can be operated in two modes, "embedded mode" and "daemon mode". The above configuration uses "daemon mode" which is the recommended mode for running Webware applications, even if "embedded mode" is the default mode for historical reasons. The configuration creates one "process group" called "webware". You can adapt and optimize the configuration by setting various options, like this:

```
WSGIDaemonProcess webware \
user=www-data group=www-data \
threads=15 \
python-home=path-to-virtual-env \
display-name='%{GROUP}' \
lang='de_DE.UTF-8' locale='de_DE.UTF-8' \
queue-timeout=45 socket-timeout=60 connect-timeout=15 \
request-timeout=60 inactivity-timeout=0 startup-timeout=15 \
deadlock-timeout=60 graceful-timeout=15 eviction-timeout=0 \
restart-interval=0 shutdown-timeout=5 maximum-requests=0
```

You can also define more than one process group, and use different process groups for different applications. In this case, mod_macro can be useful to avoid specifying the same options multiple times. It can be used like this to define different groups with a different number of threads that are created in each daemon process:

```
<Macro WSGIProcess $name $threads>
    WSGIDaemonProcess $name \
    user=www-data group=www-data \
    threads=$threads \
    display-name='%{GROUP}' \
    python-home=path-to-common-virtual-env \
    lang='de_DE.UTF-8' locale='de_DE.UTF-8' \
    queue-timeout=45 socket-timeout=60 connect-timeout=15 \
    request-timeout=60 inactivity-timeout=0 startup-timeout=15 \
    deadlock-timeout=60 graceful-timeout=15 eviction-timeout=0 \
    restart-interval=0 shutdown-timeout=5 maximum-requests=0
</Macro>
```

(continues on next page)

(continued from previous page)

```
Use WSGIProcess app1 25

WSGIScriptAlias /app1 \
    path-to-app1-work-dir/Scripts/WSGIScript.py process-group=app1

<Directory path-to-app1-work-dir/Scripts>
    Require all granted
</Directory>

Use WSGIProcess app2 10

WSGIScriptAlias /app2 \
    path-to-app2-work-dir/Scripts/WSGIScript.py process-group=app2

<Directory path-to-app2-work-dir/Scripts>
    Require all granted
</Directory>
```

In the above configurations, we are running only one process per process group, but multiple threads. The first app will use 25 threads, while the second app will use only 10. The WSGI environment variable `wsgi.multithread` will be set to `True`, while `wsgi.multiprocess` will be set to `False`. You can check these settings in your Webware application. The ThreadedAppServer of the legacy Webware for Python 2 used the same single process, multiple threads model, and is the recommended, tried and tested way to run Webware applications. But with Webware for Python 3, you can also configure mod_wsgi and other WSGI servers to run Webware applications using multiple processes, each using one or more threads. This may achieve better performance if you have many requests and your application is CPU-bound, because the GIL in Python prevents CPU-bound threads from executing in parallel. For typical I/O-bound web application, which spend most of their time waiting for the database, this is usually not a big problem. For certain applications you may want to try out the multi process model, but you need to be aware of special precautions and limitations that must be considered in this case. See the section *Caveats of Multiprocessing Mode* below and the section on processes and threading in the mod_wsgi documentation.

If you want to restart the daemon process after deploying a new version of the Webware application to the application working directory, you can do so by changing (touching) the WSGI file:

```
touch Scripts/WSGIScript.py
```

The mod_wsgi documentation also explains how to restart daemon processes by sending a *SIGINT* signal, which can be also done by the Webware application itself, and it also explains how you can monitor your application for code changes and automatically restart in that case.

10.6 Other WSGI servers

Depending on your production environment and the type of your application, it may make sense to deploy Webware applications with other WSGI servers. In the following we will give some advice for configuring some of the more popular WSGI servers to run Webware applications.

10.6.1 Using Bjoern as WSGI server

Bjoern is a fast, lightweight WSGI server for Python, written in C using Marc Lehmann's high performance libev event loop and Ryan Dahl's http-parser.

You first need to install `libev4` and `libev-devel`, then you can `pip install bjoern` into the virtual environment where you already installed Webware.

In order to make use of Bjoern, you need to add the following at the end of the `Scripts\WSGIScript.py` file in the application working directory:

```
from bjoern import run

run(application, 'localhost', 8088)
```

Since Bjoern does not support the WSGI `write()` callable, you must configure Webware to not use this mechanism, by using the following settings at the top of the `Scripts\WSGIScript.py`:

```
settings = {'WSGIWrite': False}
```

A systemd unit file at `/etc/systemd/system/bjoern.service` could look like this:

```
[Unit]
Description=Bjoern WSGI server running Webware application
After=network.target
StartLimitIntervalSec=0

[Install]
WantedBy=multi-user.target

[Service]
User=www-data
Group=www-data
PermissionsStartOnly=true
WorkingDirectory=path-to-app-work-dir
ExecStart=path-to-virtual-env/bin/python Scripts/WSGIScript.py
TimeoutSec=600
Restart=on-failure
RuntimeDirectoryMode=775
```

You can then enable and run the service as follows:

```
systemctl enable bjoern
systemctl start bjoern
```

10.6.2 Using MeinHeld as WSGI server

MeinHeld is another lightweight, high performance WSGI server.

You first need to `pip install meinheld` into the virtual environment where you already installed Webware.

Add the following at the end of the `Scripts\WSGIScript.py` file in the application working directory in order to use MeinHeld:

```
from meinheld import server

server.listen(("127.0.0.1", 8080))
server.run(application)
```

Similarly to Bjoern, you need to also adapt the settings at the top of the `Scripts\WSGIScript.py` file:

```
settings = {'WSGIWrite': False}
```

10.6.3 Using CherryPy as WSGI server

Cherrypy is a minimalist Python web framework that also contains a reliable, HTTP/1.1-compliant, WSGI thread-pooled webserver.

TO make use of CherryPy's WSGI server, add the following at the end of the `Scripts\WSGIScript.py` file in the application working directory:

```
import cherrypy

cherrypy.tree.graft(application, '/')
cherrypy.server.unsubscribe()
server = cherrypy._cpserver.Server()
server.socket_host = '127.0.0.1'
server.socket_port = 8080
server.thread_pool = 30
server.subscribe()
cherrypy.engine.start()
cherrypy.engine.block()
```

10.6.4 Using uWSGI as WSGI server

The uWSGI project aims at developing a full stack for building hosting services, and it also contains a WSGI server component.

You first need to `pip install uwsgi` into the virtual environment where you already installed Webware.

You can start the uWSGI server component as follows:

```
cd path-to-app-work-dir
. ../.venv/bin/activate
uwsgi --http-socket 127.0.0.1:8080 --threads 30 \\
    --virtualenv path-to-virtual-env --wsgi-file Scripts/WSGIScript.py
```

You can also create a systemd file to run this automatically when the system boots, as explained above.

Many more uWSGI configuration options are explained in the uWSGI documentation, we will not go into more details here.

10.6.5 Using Gunicorn as WSGI server

Gunicorn is a fast WSGI server for Unix using a pre-fork worker model.

You first need to `pip install gunicorn` into the virtual environment where you already installed Webware.

You can start the Gunicorn WSGI server as follows:

```
cd path-to-app-work-dir
. ../.venv/bin/activate
PYTHONPATH=Scripts gunicorn -b 127.0.0.1:8080 WSGIScript:application
```

You can also create a systemd file to run this automatically when the system boots, as explained above.

Many more Gunicorn configuration options are explained in the Gunicorn documentation, we will not go into more details here.

10.7 Sourceless Installs

When deploying a Webware application, you do not really need to copy the source code to the production system, it suffices to deploy the compiled compiled Python files. Though this is actually not considered a good practice, and it also does not really help to keep the source code secret (as it can be decompiled pretty easily), there may be reasons why you still want to do this, for instance to impede casual users to tinker with your code on the server.

To do this, you first need to compile all your Python files in the application working directory:

```
cd path-to-app-work-dir
. ../.venv/bin/activate
python -OO -m compileall -b .
```

By activating the virtual environment, you make sure that you compile the source files with the proper Python version. The `-b` option puts the compiled files as siblings to the source files using the `.pyc` extension, which is essential here. The `-OO` option removes all assert statements and docstrings from the code.

If you want to serve contexts outside the application working directory, like the default Examples or Admin context, you need to compile these as well, in a similar way.

You can now remove all the source files except the WSGI script and the `__pycache__` directories, they are not needed on the production system anymore:

```
cd path-to-app-work-dir
find . -type f -name '*.py' -delete -o \
       -type d -name 'Scripts' -prune -o \
       -type d -name __pycache__ -exec rm -rf {} \+
```

In order to make this work, you will also need to modify some settings in `Configs/Application.config`, like this:

```
ExtensionsToIgnore = {
    '.py', '.pyo', '.tmpl', '.bak', '.py_bak',
    '.py~', '.psp~', '.html~', '.tmpl~'
}
```

(continues on next page)

(continued from previous page)

```
ExtensionCascadeOrder = ['.pyc', '.psp', '.html']
FilesToHide = {
    '.*', '*~', '*.bak', '*.py_bak', '*.tmpl',
     '*.py', '*.pyo', '__init__.*', '*.config'
}
```

10.8 Caveats of Multiprocessing Mode

As explained above, it is possible to operate mod_wsgi and some other WSGI servers in multiprocessing mode, where several processes serve the same Webware application in parallel, or you can run several multithreaded WSGI servers in parallel, maybe even on different machines, and use a load balancer as a reverse proxy to distribute the load between the different servers.

This is totally doable, and may make sense in order to better utilize existing hardware. Because of the the GIL, a multithreaded Python application will not be able to get the full performance from a multi-core machine when running a CPU-bound application. However, there are some caveats that you need to be aware of:

- The Webware TaskManager will be started with every Application process. If this is not what you want, you can change the `RunTasks` configuration setting to False, and run the TaskManager in a dedicated process.
- Some load balancers support "sticky sessions", identifying clients by their session cookies and dispatching them to the same server processes. But usually, in multiprocessing mode, you cannot guarantee that requests from the same client are served by the same process, and it would also partially defeat the whole purpose of running multiple processes. Therefore, the SessionMemoryStore, SessionFileStore and SessionDynamicStore are not suitable in that mode, since the session data that is created in the local memory of one process will not be available in a different process. Also, accessing session files from different processes simultaneously can be problematic. Instead, we recommend changing the `SessionStore` configuration setting to use the SessionRedisStore or the SessionMemcachedStore. Storing the session data in the database is also possible, but may degrade performance.
- When caching frequently used data in local memory, this will become less effective and waste memory when running multiple processes. Consider using a distributed caching system such as Redis or Memcached instead. If you are using the SessionRedisStore or the SessionMemcachedStore, you will need to install one of these systems anyway.
- Webware applications often store global, application wide state in class attributes of servlet classes or elsewhere in local memory. Again, be aware that this does not work anymore if you are running the same application in multiple processes.
- Redirecting standard error and output to the same log file is not supported when running multiple processes, so the `LogFilename` setting should be set to None, and a different logging mechanism should be used. When using mod_wsgi you may need to use the `WSGIRestrictStdout` directive and log on that level. Future versions of Webware for Python 3 will address this problem and provide proper logging mechanisms instead of just printing to stdout.

CHAPTER

ELEVEN

PLUG-INS

Webware for Python supports "plug-ins" to extend the framework and provide additional capabilities.

In Webware for Python 3, plug-ins are implemented as packages with metadata ("entry points") through which they can be automatically discovered, even if they have been installed independetly of Webware. You only need to specify which plug-ins shall be loaded in the `PlugIns` configuration setting, and Webware will automatically load them if they are installed.

Every Webware plug-in is a Python package, i.e. a directory that contains a `__init__.py` file and optionally other files. As a Webware plugin, it must also contain a special `Properties.py` file. You can disable a specific plug-in by placing a `dontload` file in its package directory.

If you want to distribute a Webware plug-in, you should advertize it as an entry point using the `webware.plugins` identifier in the `setup.py` file used to install the plug-in.

The `__init.py__` file of the plug-in must contain at least a function like this:

```
def installInWebware(application):
    pass
```

The function doesn't need to do anything, but this gives it the opportunity to do something with the global Webware `Application` object. For instance, the PSP plugin uses `addServletFactory.addServletFactory` to add a handler for `.psp` files.

The `Properties.py` file should contain a number of assignments:

```
name = "Plugin name"
version = (1, 0, 0)
status = 'beta'
requiredPyVersion = (3, 6)
requiredOpSys = 'posix'
synopsis = """A paragraph-long description of the plugin"""
webwareConfig = {
    'examplePages': [
        'Example1',
        'ComplexExample',
        ]
    }
def willRunFunc():
    if softwareNotInstalled:
        return "some message to that effect"
    else:
        return None
```

If you want to provide some examples for using your plug-in, they should be put in an `Examples/` subdirectory.

A plugin who's `requiredPyVersion` or `requiredOpSys` aren't satisfied will simply be ignored. `requiredOpSys` should be something returned by `os.name`, like `posix` or `nt`. Or you can define a function `willRunFunc` to test. If there aren't requirements you can leave these variables and functions out.

If you plan to write your own Webware plug-in, also have a look at our *Style Guidelines* and the source code of the built-in plug-ins (PSP, TaskKit, UserKit, WebUtils, MiscUtils) which can serve as examples. We also recommend to add some tests to your plug-in, see the section on *Testing*.

CHAPTER

TWELVE

STYLE GUIDELINES

12.1 Introduction

Style refers to various aspects of coding including indentation practices, naming conventions, use of design patterns, treatment of constants, etc. One of the goals of Webware war to maintain fairly consistent coding style with respect to certain basics as described in this document.

This document is therefore important for those who develop Webware itself or who wish to contribute code, although ordinary users may still find it interesting and useful in understanding the Webware APIs.

Please keep in mind that Webware for Python was developed when modern Python features like properties and style guidelines such as PEP8 did not yet exist. Therefore the API and code style used in Webware for Python may look a bit antiquated today. However, we decided to keep the old API in Webware for Python 3, and still follow most of the original style guidelines, in order to stay backward compatible and make migration of existing Webware for Python apps as painless as possible.

12.2 Syntax and Naming

12.2.1 Methods and Attributes

Methods and attributes are an important topic because they are used so frequently and therefore have an impact on using the classes, learning them, remembering them, etc.

The first thing that is important to understand is that Webware is constructed in an object-oriented fashion, including full encapsulation of object attributes. In other words, you communicate and use objects completely via their methods (except that classes and subclasses access their own attributes &ndash; somebody's got to).

The primary advantages of using methods are:

- Method implementations can be changed with minimal impact on the users of the class.
- Subclasses can customize methods to suit their needs and still be used in the usual fashion (as defined by the superclass).

In the case of a method that returns a value, that value may manifest in several ways:

1. Stored as attribute.
2. Stored as an attribute, only on demand (e.g., lazy and cached).
3. Computed upon every request.
4. Delegated to another object.

By requiring that object access is done solely through methods, the implementer of the class is free to switch between the techniques above.

Keeping that in mind, it is apparent that naming conventions are needed for attributes, the methods that return them and the methods that set them. Let's suppose the basic "thing" is `status`. The convention then is:

- `_status` - the attribute
- `status()` - the retrieval method
- `setStatus()` - the setter method

The underscore that precedes the attribute denotes that it is semi-private and allows for a cleanly named retrieval method. The `status()` and `setStatus()` convention originated many years ago with Smalltalk and proved successful with that language as well as others, including Objective-C.

Some styles prefix the retrieval method with `get`, but Webware does not for the sake of brevity and because there are methods for which it is not always clear if a `get` prefix would make sense.

Methods that return a Boolean are prefixed with `is` or `has` to make their semantics more obvious. Examples include `request.isSecure()`, `user.isActive()` and `response.hasHeader()`.

12.2.2 Method Categories

Webware classes divide their methods into logical groups called categories purely for organizational purposes. This often helps in understanding the interface of a class, especially when looking at its summary.

Upon installation, Webware generates HTML summaries for each Python source file. See the summary of HTTPResponse for an example.

By convention, a category is named with a comment beginning with *# region* and ending with *# endregion*. IDEs such as PyCharm will recognize these sections as foldable blocks.

12.2.3 Abbreviations

Using abbreviations is a common coding practice to reduce typing and make lines shorter. However, removing random characters to abbreviate is a poor way to go about this. For example, `transaction` could be abbreviated as `trnsact` or `trnsactn`, but which letters are left out is not obvious or easy to remember.

The typical technique in Webware is to use the first whole syllable of the word. For example, `trans` is easy to remember, pronounce and type.

Attributes and methods that return the number of some kind of object are quite common. Suppose that the objects in questions are requests. Possible styles include `numRequests`, `numberOfRequests`, `requestCount` and so on. Webware uses the first style in all cases, for consistency:

```
numRequests
```

If there is a corresponding attribute, it should have the same name (prefixed by an underscore).

12.2.4 Compound Names

Identifiers often consist of multiple names. There are three major styles for handling compound names:

1. `serverSidePath` - the Webware convention
2. `serversidepath`
3. `server_side_path`

Python itself used all three styles in the past (`isSet`, `getattr`, `has_key`), but Webware uses only the first which is more readable than the second and easier to type that the third.

This rule also applies to class names such as `HTTPRequest` and `ServletFactory`.

12.2.5 Component Names

Names of object-oriented Webware components are often suffixed with **Kit**, as in **UserKit** and **MiddleKit**.

The occasional component that serves as a collective library (rather than an OO framework) is suffixed with **Utils**, as in *MiscUtils* and *WebUtils*.

12.2.6 Rules

We follow PEP 8 and usual Python conventions with these exceptions (for historical reasons and to remain backward compatible):

- Filenames are capitalized
- Method names are camelCase
- Attributes start with an underscore
- Getters and Setters for these attributes are ordinary methods.
- Setters use a "set" prefix, but getters do nt use a "get" prefix.

12.3 Structure and Patterns

12.3.1 Classes

Webware overwhelmingly uses classes rather than collections of functions for several reasons:

- Classes allow for subclassing and therefore, a proven method of software extension and customization.
- Classes allow for creating multiple instances (or objects) to be used in various ways. Examples include caching created objects for increased performance and creating multi-threaded servers.
- Classes can provide for and encourage encapsulation of data, which gives implementers more freedom to improve their software without breaking what depends on it.

By using classes, all three options above are available to the developer on an on-going basis. By using collections of functions, none of the above are readily available.

Note that making classes in Python is extraordinarily easy. Doing so requires one line:

```python
class Foo(SuperFoo):
```

and the use of `self` when accessing attributes and methods. The difference in time between creating a class with several methods vs. a set of several functions is essentially zero.

12.3.2 Configuration Files

Specific numbers and strings often appear in source code for determining things such as the size of a cache, the duration before timeout, the name of the server and so on. Rather than place these values directly in source, Webware provides configuration files that are easily discerned and edited by users, without requiring a walk through the source.

Webware uses ordinary Python dictionaries for configuration files for several reasons:

- Those who know Python will already understand the syntax.
- Python dictionaries can be quickly and easily read (via Python's `eval()` function).
- Dictionaries can contain nested structures such as lists and other dictionaries, providing a richer configuration language.

By convention, these configuration files:

- Contain a Python dictionary.
- Use a `.config` extension.
- Capitalize their keys.
- Are named after the class that reads them.
- Are located in a `Configs/` subdirectory or in the same directory as the program.

Webware provides a `Configurable` mix-in class that is used to read configuration files. It allows subclasses to say `self.setting('Something')` so that the use of configuration information is easy and recognizable throughout the code.

12.3.3 Accessing Named Objects

Several classes in Webware store dictionaries of objects keyed by their name. `HTTPRequest` is one such class which stores a dictionary of form fields. The convention for providing an interface to this information is as follows:

```
# region Fields
def field(self, name, default=_NoDefault):
def hasField(self, name):
def fields(self):
```

A typical use would be:

```
req.field('city')
```

which returns the field value of the given name or raises an exception if there is none. Like the `get()` method of Python's dictionary type, a second parameter can be specified which will be returned if the value is not found:

```
req.field('city', None)
```

`req.hasField('city')` is a convenience method that returns `True` if the value exists, `False` otherwise.

`req.fields()` returns the entire dictionary so that users have complete access to the full suite of dictionary methods such as `keys()`, `values()`, `items()`, etc. Users of this method are trusted not to modify the dictionary in any way. A paranoid class may choose to return a copy of the dictionary to help reduce abuse (although Webware classes normally do not for performance reasons).

In cases where the user of the class should be able to modify the named objects, the following methods are provided:

```
def setValue(self, name, value):
def delValue(self, name):
```

Often Python programmers will provide dictionary-style access to their objects by implementing `__getitem__` so that users may say:

```
req['city']
```

Webware generally avoids this approach for two reasons. The first and most important is that Webware classes often have more than one set of named objects. For example, HTTPRequest has both fields and cookies. HTTPResponse has both cookies and headers. These objects have their own namespaces, making the semantics of `obj['name']` ambiguous. Also, the occasional class that has only one dictionary could potentially have more in the future.

The second reason is the readability provided by expressions such as `response.cookie(name)` which states clearly what is being asked for.

In those cases where objects provide dictionary-like access, the class is typically a lightweight container that is naturally thought of in terms of its dictionary components. Usually these classes inherit from `dict`.

12.3.4 Components

Webware consists of multiple components that follow particular conventions, not only for the sake of consistency, but also to enable scripts to manipulate them (such as generating documentation upon installation).

Example components include PSP, TaskKit and MiscUtils.

These conventions are not yet formally documented, however if you quickly browse through a couple components, some conventions about directory structure and source code become apparent.

Also, if a component serves as a Webware plug-in, then there are additional conventions for them to follow in order to work correctly. See the chapter on *Plug-ins* for details.

12.3.5 Breaking the Rules

Of course, there are times when the rules are broken for good reason. To quote a cliché: "Part of being an expert is knowing when to break the rules."

But regarding style, Webware developers do this very infrequently for the reasons stated in the introduction.

CHAPTER

THIRTEEN

PSP

13.1 Summary

Python Server Pages (PSP) provides the capability for producing dynamic web pages for use with the Webware Python Servlet engine simply by writing standard HTML. The HTML code is interspersed with special tags that indicate special actions that should be taken when the page is served. The general syntax for PSP has been based on the popular Java Server Pages specification used with the Java Servlet framework.

Since the Webware Servlets are analogous to Java Servlets, PSP provides a scripting language for use with it that includes all of the power of Python. You will find that PSP compares favorably to other server side web scripting languages, such as ASP, PHP and JSP.

Features of PSP include:

- Familiar Syntax (ASP, JSP, PHP)
- The power of Python as the scripting language
- Full access to the Webware Servlet API
- Flexible PSP Base Class framework
- Ability to add additional methods to the class produced by PSP

13.2 Feedback

The PSP for Webware project is fully open source. Help in all areas is encouraged and appreciated. Comments should be directed to the Webware Discussion mailing list. This is a relatively low volume list and you are encouraged to join the list if you wish to participate in the development of PSP or Webware, or if you plan on developing an application using the framework.

13.3 General Overview

The general process for creating PSP files is similar to creating an HTML page. Simply create a standard HTML page, interspersed with the special PSP tags that your needs require. The file should be saved with an extension of `.psp`. Place this file in a directory that is served by Webware. When a request comes in for this page, it will be dynamically compiled into a Webware servlet class, and an instance of this class will be instantiated to serve requests for that page.

There are two general types of PSP tags, `<%...%>` and `<psp:...>`. Each of these tags have special characteristics, described below.

Whether or not you will need to include standard HTML tags in the start of your PSP page, such as `<html>`, `<head>` etc. depends on the base class you choose for your PSP class. The default setup does not output any of those tags automatically.

13.4 PSP Tags

The following tags are supported by the current PSP implementation.

13.4.1 Expression Tag – `<%= expression %>`

The expression tag simply evaluates some Python code and inserts its text representation into the HTML response. You may include anything that will evaluate to a value that can be represented as a string inside the tag.

Example:

```
The current time is <%= time.time() >
```

When the PSP parsing engine encounters Expression tags, it wraps the contents in a call to the Python `str()` function. Multiple lines are not supported in a PSP expression tag.

13.4.2 Script Tag – `<% script code %>`

The script tag is used to enclose Python code that should be run by the Webware Servlet runner when requests are processed by the Servlet which this PSP page produces. Any valid Python code can be used in Script tags. Inside a script tag, indentation is up to the author, and is used just like in regular Python (more info on blocks below). The PSP Engine actually just outputs the strings in a Script tag into the method body that is being produced by this PSP page.

Example:

```
<% for i in range(5):
    res.write("<b>This is number" + str(i) + "</b><br>") %>
```

The Python code within script tags has access to all local and class variables declared in the PSP page, as well as to all variables of the enclosing class of this PSP page.

Special local variables that will be available in all PSP pages are:

`req`
: The HTTRequest object for this page.

`res`
: The HTTPResponse object for this page. The HTTPResponse object includes the *write* method that is used to output HTML to the client.

`trans`
: The Transaction object for this client request. The Transaction object provides access to the objects involved in servicing this client request.

Python Code Blocks that span PSP Script Tags

The Python code structure, which uses whitespace to signify blocks of code, presents a special challenge in PSP pages. In order to allow for readable HTML code that does not impose restrictions on straight HTML within PSP pages, PSP uses a special syntax to handle Python blocks that span script tags.

Automatic Blocks

Any script tag with Python code that ends with a colon (:) is considered to begin a block. A comment tag may follow the colon. After this tag, any following HTML is considered to be part of the block begun within the previous script tag. To end the block, insert a new script tag with the word "end" as the only statement.

Example of Script/HTML block:

```
<% for i in range(5): %>  # the blocks starts here, no need for indenting the following␣
↪HTML
<tr><td><%= i%></td></tr>
<% end %> The "end" statement ends the block
```

These blocks can be nested, with no need for special indentation, and each script tag that only contains a solitary end statement will reduce the block indent by one.

Manual Blocks

It is also possible to force a block of HTML statements to be included in a block. You might want to do this if your start a loop of some kind in a script tag, but need the first line of the loop to also be inside the script tag. In this case, the automatic indenting described above wouldn't notice the block, because the last line in the script tag wouldn't be a ":". In this case, you need to end the script tag with `$%>`. When a script tag ends with `$%>`, the PSP Parser will indent the following HTML at the same level as the last line of the script tag. To end this level of indentation, just start another script tag. Easy.

Example of Manual Indention Script/HTML block:

```
<% for i in range(5):
    icubed = i*i $%>  # The following lines of straight HTML will be included in the␣
↪same block this line is on
  <tr><td><%= icubed%></td></tr>
<% pass %>  # Any new script statement resets the HTML indentation level
```

You could also start a new script block that just continues at the same indentation level that the HTML and the previous script block were at.

Braces

PSP also supports using braces to handle indentation. This goes against the grain of Python, we know, but is useful for this specific application. To use this feature, specify it as you indentation style in a page directive, like so:

```
<%@page indentType="braces" %>
```

Now use braces to signify the start and end of blocks. The braces can span multiple script tags. No automatic indentation will occur. However, you must use braces for all blocks! Tabs and spaces at the start of lines will be ignored and removed!

Example:

```
This is <i>Straight HTML</i><br>
<%
  for i in range(5): { %>  # Now I'm starting a block for this loop
  z = i*i
%>
<!-- Now I'm ending the scripting tag that started the block,
but the following lines are still in the block -->
More straight HTML. But this is inside the loop started above.<br>
My i value is now <%= i %><br>
Now I will process it again.<br>
<%
  v = z*z
%>
Now it is <%=v %>
<% } %>  # End the block
```

13.4.3 File and Class Level Code – `<psp:file>` and `<psp:class>`

The file and class level script tag allows you to write Python code at the file (module) level or class level. For example, at the file level, you might do imports statements, and some initialization that occurs only when the PSP file is loaded the first time. You can even define other classes that are used in your PSP file.

Example:

```
<psp:file>
  # Since this is at the module level, _log is only defined once for this file
  import logging
  _log = logging.getLogger( __name__ )
</psp:file>
<html>
  <% _log.debug('Okay, Ive been called.') %>
  <p>Write stuff here.</p>
</html>
```

At the class level you can define methods using ordinary Python syntax instead of the <psp:method > syntax below.

Example:

```
<psp:class>
  def writeNavBar(self):
    for uri, title in self.menuPages():
      self.write( "<a href="%s">%s</a>" % (uri, title) )
</psp:class>
```

Indentation is adjusted within the file and class blocks. Just make your indentation consistent with the block, and PSP will adjust the whole block to be properly indented for either the class or the file. For example file level Python would normally have no indentation. But in PSP pages, you might want some indentation to show it is inside of the `<psp:file>...</psp:file>` tags. That is no problem, PSP will adjust accordingly.

There is one special case with adding methods via the `<psp:class>` tag. The `awake()` method requires special handling, so you should always use the `<psp:method>` tag below if you want to override the `awake()` method.

13.4.4 Method Tag – `<psp:method ...>`

The Method tag is used to declare new methods of the Servlet class this page is producing. It will generally be more effective to place method declarations in a Servlet class and then have the PSP page inherit from that class, but this tag is here for quick methods. The Method tag may also be useful for over-riding the default functionality of a base class method, as opposed to creating a Servlet class with only a slight change from another.

The syntax for PSP methods is a little different from that of other tags. The PSP Method declaration uses a compound tag. There is a beginning tag `<psp:method name="methname" params="param1, param2">` that designates the start of the method definition and defines the method name and the names of its parameters. The text following this tag is the actual Python code for the method. This is standard Python code, with indentation used to mark blocks and no raw HTML support. It is not necessary to start the method definition with indentation, the first level of indention is provided by PSP.

To end the method definition block, the close tag `</psp:method>` is used.

Example:

```
<psp:method name="MyClassMethod" params="var1, var2">
  import string
  return string.join((var1,var2),'')
</psp:method>
```

This is a silly function that just joins two strings. Please note that it is not necessary to declare the self parameter as one of the function’s parameters. This will be done automatically in the code that PSP generates.

A PSP:Method can be declared anywhere in the psp sourcefile and will be available throughout the PSP file through the standard `self.PSPMethodName(parameters)` syntax.

13.4.5 Include Tag – `<psp:include ...>`

The include tag pauses processing on the page and immediately passes the request on the the specified URL. The output of that URL will be inserted into the output stream, and then processing will continue on the original page. The main parameter is `path`, which should be set to the path to the resources to be included. This will be relative to the current page, unless the path is specified as absolute by having the first character as “/”. The path parameter can point to any valid url on this Webware application. This functionality is accomplished using the Webware Application’s forwardRequestFast function, which means that the current Request, Response and Session objects will also be used by the URL to which this request is sent.

Example:

```
<psp:include path="myfile.html">
```

13.4.6 Insert Tag – `<psp:insert ...>`

The insert tag inserts a file into the output stream that the psp page will produce, but does not parse that included file for psp content. The main parameter is `file`, which should be set to the filename to be inserted. If the filename starts with a “/”, it is assumed to be an absolute path. If it doesn’t start with a “/”, the file path is assumed to be relative to the psp file. The contents of the insert file will not be escaped in any way except for triple-double-quotes (&quot;&quot;&quot;), which will be escaped.

At every request of this servlet, this file will be read from disk and sent along with the rest of the output of the page.

This tag accepts one additional parameter, “static”, which can be set to “True” or “1”. Setting this attribute to True will cause the inserted file’s contents to be embedded in the PSP class at generation time. Any subsequent changes to the file will not be seen by the servlet.

Example:

```
<psp:insert file="myfile.html">
```

13.5 Directives

Directives are not output into the HTML output, but instead tell the PSP parser to do something special. Directives have at least two elements, the type of directive, and one or more parameters in the form of `param="value"` pairs.

Supported Directives include:

13.5.1 Page Directive – `<%@ page ... %>`

The page directive tells the PSP parser about special requirements of this page, or sets some optional output options for this page. Directives set in `page` apply to the elements in the current PSP source file and to any included files.

Supported Page parameters:

- `imports` – The imports attribute of the page directive tells the PSP parser to import certain Python modules into the PSP class source file.

  The format of this directive is as follows:

  Example: `<%@ page imports="sys,os"%>`

  The value of the imports parameter may have multiple, comma separated items.

  `from X import Y` is supported by separating the source package from the object to be imported with a colon (:), like this:

  Example: `<%@ page imports="os:path" %>`

  This will import the path object from the os module.

  Please note the = sign used in this directive. Those who are used to Python might try to skip it.

- `extends` – The extends attribute of the page tells the PSP parser what base class this Servlet should be derived from.

  The PSP servlet produced by parsing the PSP file will inherit all of the attributes and methods of the base class.

  The Servlet will have access to each of those attributes and methods. They will still need to be accessed using the "self" syntax of Python.

  Example: `<%@ page extends="MyPSPBaseClass"%>`

  This is a very powerful feature of PSP and Webware. The developer can code a series of Servlets that have common functionality for a series of pages, and then use PSP and the extends attribute to change only the pieces of that base servlet that are specific to a certain page. In conjunction with the `method` page attribute, described below, and/or the `<psp:method ...>` tag, entire sites can be based on a few custom PSP base classes. The default base class is `PSPPage.py`, which is inherited from the standard Webware `Page.py` servlet.

  You can also have your PSP inherit from multiple base classes. To do this, separate the base classes using commas, for example `<%@ page extends="BaseClass1,BaseClass2"%>`. If you use a base class in `<%@ page extends="..."%>` that is not specifically imported in a `<%@ page imports="..."%>` directive, the base class will be assumed to follow the servlet convention of being in a file of the same name as the base class plus the ".py" extension.

- `method` – The `method` attribute of the `page` directive tells the PSP parser which method of the base class the HTML of this PSP page should be placed in and override.

  Example: `<%@ page method="WriteHTML"%>`

  Standard methods are `WriteHTML`, of the standard `HTTPServlet` class, and `writeBody`, of the `Page` and `PSPPage` classes. The default is `writeBody`. However, depending on the base class you choose for your PSP class, you may want to override some other method.

- `isThreadSafe` – The `isThreadSafe` attribute of `page` tells the PSP parser whether the class it is producing can be utilized by multiple threads simultaneously. This is analogous to the `isThreadSafe` function in Webware servlets.

  Example: `<%@ page isThreadSafe="yes"%>`

  Valid values are "yes" and "no". The default is "no".

- `isInstanceSafe` – The `isInstanceSafe` attribute of the `page` directive tells the PSP parser whether one instance of the class being produced may be used multiple times. This is analogous to the isInstanceSafe function of Webware Servlets.

  Example: `<%@ page isInstanceSafe="yes"%>`

  Valid values are "yes" and "no". The default is "yes".

- `indentType` – The `indentType` attribute of the `page` directive tells the parser how to handle block indention in the Python sourcefile it creates. The `indentType` attribute sets whether the sourcefile will be indented with tabs or spaces, or braces. Valid values are "tabs", "spaces" or "braces". If this is set to "spaces", see `indentSpaces` for setting the number of spaces to be used (also, see blocks, below). The default is "spaces".

  Example: `<%@ page indentType="tabs"%>`

  This is a bit of a tricky item, because many editors will automatically replace tabs with spaces in their output, without the user realizing it. If you are having trouble with complex blocks, look at that first.

- `indentSpaces` – Sets the number of spaces to be used for indentation when `indentType` is set to spaces. The default is "4".

  Example: `<%@ page indentSpaces="8" %>`

- `gobbleWhitespace` – The `gobblewhitespace` attribute of the `page` directive tells the PSP parser whether it can safely assume that whitespace characters that it finds between two script tags can be safely ignored. This is a special case directive. It applies when there are two script tags of some kind, and there is only whitespace characters between the two tags. If there is only whitespace, the parser will ignore the whitespace. This is necessary for multipart blocks to function correctly. For example, if you are writing an if/else block, you would have your first script block that starts the if, and then you would end that block and start a new script block that contains the else portion. If there is any whitespace between these two script blocks, and `gobbleWhitespace` is turned off, then the parser will add a write statement between the two blocks to output the whitespace into the page. The problem is that the write statement will have the indentation level of the start of the if block. So when the else statement starts, it will not be properly associated with the preceding if, and you'll get an error.

  If you do need whitespace between two script blocks, use the `&nbsp;` code.

  Example: `<%@ page gobbleWhitspace="No"%>`

  Valid values are "yes" and "no". The default is "yes".

- `formatter` – The `formatter` attribute of the `page` directive can be used to specify an alternative formatter function for `<%= ... %>` expression blocks. The default value is `str`. You might want to use this if certain types need to be formatted in a certain way across an entire page; for example, if you want all integers to be formatted with commas like "1,234,567" you could make that happen by specifying a custom formatter.

  Example:

```
<%@ page imports="MyUtils" %>
<%@ page formatter="MyUtils.myFormatter" %>
```

13.5.2 Include Directive – `<%@ include ... %>`

The include directive tells the parser to insert another file at this point in the page and to parse it for psp content. It is generally no problem to include an html file this way. However, if you do not want your include file to be parsed, you may use the `<psp:include ...>` tag described above.

Example:

```
<%@ include file="myfile.txt"%>
```

13.6 Other Tags

- **Declaration** (`<%! ... %>`) – No need for this tag. Simply use script tags to declare local variables.
- **Forwarding** functionality is now available in Webware, but no tag based support has been added to PSP yet.

13.7 Developers

The original author of PSP is Jay Love and the project was later maintained by Jay and Geoff Talvola. The contributions of the entire Webware community have been invaluable in improving this software.

Some architectural aspects of PSP were inspired by the Jakarta Project.

CHAPTER

FOURTEEN

USERKIT

UserKit provides for the management of users including passwords, user data, server-side archiving and caching. Users can be persisted on the server side via files or the external MiddleKit plug-in.

14.1 Introduction

UserKit is a self contained library and is generally not dependent on the rest of Webware. It does use a few functions in MiscUtils. The objects of interest in UserKit are Users, UserMangers, and Roles.

`User` – This represents a particular user and has a name, password, and various flags like `user.isActive()`.

`UserManager` – Your application will create one instance of a UserManager and use it to create and retrieve Users by name. The UserManager comes in several flavors depending on support for Roles, and where user data is stored. For storage, UserManagers can save the user records to either a flat file or a MiddleKit store. Also user managers may support Roles or not. If you don't need any roles and want the simplest UserManager, choose the UserManagerToFile which saves its data to a file. If you want hierarchical roles and persistence to MiddleKit, choose RoleUserManager-ToMiddleKit.

`Role` – A role represents a permission that users may be granted. A user may belong to several roles, and this is queried using the method `roleUser.playsRole(role)`. Roles can be hierarchical. For example a customers role may indicate permissions that customers have. A staff role may include the customers role, meaning that members of staff may also do anything that customers can do.

14.2 Examples and More Details

The docstrings in *`UserManager`* is the first place to start. It describes all the methods in *`UserManager`*. Then go to the file `Tests/TestExample.py` which demonstrates how to create users, log them in, and see if they are members of a particular role.

Once you get the idea, the docstrings in the various files may be perused for more details. See also the *reference documentation* for an overview of the available classes and methods.

14.3 Encryption of Passwords

Generally one should never save users' passwords anywhere in plain text. However UserKit intentionally does no support encryption of passwords. That is left to how you use UserKit in your application. See `TestExample.py`, for a demonstration of how easy this is using SHA digests to encrypt passwords. Basically you encrypt your password before you give it to UserKit. It is as simple as this:

```
usermanager.createUser('johndoe', sha('buster').hexdigest())
```

This design decision is to decouple UserKit from your particular encryption requirements, and allows you to use more advanced algorithms as they become available.

14.4 Credit

Author: Chuck Esterbrook, and a cast of dozens of volunteers. Thanks to Tom Schwaller for design help.

CHAPTER

FIFTEEN

TASKKIT

TaskKit provides a framework for the scheduling and management of tasks which can be triggered periodically or at specific times. Tasks can also be forced to execute immediately, set on hold or rescheduled with a different period (even dynamically).

To understand how TaskKit works, please read the following quick start article and have a look at the *reference documentation*. Also, in the "Task" subdirectory of Webware, you will find a real world use of this kit.

15.1 Scheduling with Python and Webware

The Webware for Python web application framework comes with a scheduling plug-in called TaskKit. This quick start guide describes how to use it in your daily work with Webware and also with normal Python programs (slightly updated version of an article contributed by Tom Schwaller in March 2001).

Scheduling periodic tasks is a very common activity for users of a modern operating system. System administrators for example know very well how to start new `cron` jobs or the corresponding Windows analogues. So, why does a web application framework like Webware need its own scheduling framework? The answer is simple: Because it knows better how to react to a failed job, has access to internal data structures, which otherwise would have to be exposed to the outside world and last but not least it needs scheduling capabilities anyway (e.g. for session sweeping and other memory cleaning operations).

Webware is developed with the object oriented scripting language Python, so it seemed natural to write a general purpose Python based scheduling framework. One could think that this problem is already solved (remember the Python slogan: batteries included), but strange enough there has not much work been done in this area. The two standard Python modules `sched.py` and `bisect.py` are way too simple, not really object oriented and also not multithreaded. This was the reason to develop a new scheduling framework, which can not only be used with Webware but also with general purpose Python programs. Unfortunately scheduling has an annoying side effect. The more you delve into the subject the more it becomes difficult.

After some test implementations I discovered the Java scheduling framework of the "Ganymede" network directory management system and took it as a model for the Python implementation. Like any other Webware plug-in the TaskKit is self contained and can be used in other Python projects. This modularity is one of the real strengths of Webware and in sharp contrast to Zope where people tend to think in Zope and not in Python terms. In a perfect world one should be able to use web wrappers (for Zope, Webware, Quixote, . . .) around clearly designed Python classes and not be forced to use one framework. Time will tell if this is just a dream or if people will reinvent the "Python wheels" over and over again.

15.2 Tasks

The TaskKit implements the three classes `Scheduler, TaskHandler` and `Task`. Let's begin with the simplest one, i.e. Task. It's an abstract base class, from which you have to derive your own task classes by overriding the `run()`-method like in the following example:

```
from time import strftime, localtime
from TaskKit.Task import Task

class SimpleTask(Task):

    def run(self):
        print self.name(), strftime("%H:%M:%S", localtime())
```

`self.name()` returns the name under which the task was registered by the scheduler. It is unique among all tasks and scheduling tasks with the same name will delete the old task with that name (so beware of that feature!). Another simple example which is used by Webware itself is found in `Tasks/SessionTask.py`:

```
from TaskKit.Task import Task

class SessionTask(Task):

    def __init__(self, sessions):
        Task.__init__(self)
        self._sessionstore = sessions

    def run(self):
        if self.proceed():
            self._sessionstore.cleanStaleSessions(self)
```

Here you see the `proceed()` method in action. It can be used by long running tasks to check if they should terminate. This is the case when the scheduler or the task itself has been stopped. The latter is achieved with a `stopTask()` call which is not recommended though. It's generally better to let the task finish and use the `unregister()` and `disable()` methods of the task handler. The first really deletes the task after termination while the second only disables its rescheduling. You can still use it afterwards. The implementation of `proceed()` is very simple:

```
def proceed(self):
    """Check whether this task should continue running.

    Should be called periodically by long tasks to check if the system
    wants them to exit. Returns True if its OK to continue, False if
    it's time to quit.

    """
    return self._handle._isRunning
```

Take a look at the `SimpleTask` class at the end of this article for an example of using `proceed()`. Another thing to remember about tasks is, that they know nothing about scheduling, how often they will run (periodically or just once) or if they are on hold. All this is managed by the task wrapper class `TaskHandler`, which will be discussed shortly. Let's look at some more examples first.

15.3 Generating static pages

On a high traffic web site (like slashdot) it's common practice to use semi-static page generation techniques. For example you can generate the entry page as a static page once per minute. During this time the content will not be completely accurate (e.g. the number of comments will certainly increase), but nobody really cares about that. The benefit is a dramatic reduction of database requests. For other pages (like older news with comments attached) it makes more sense to generate static versions on demand. This is the case when the discussion has come to an end, but somebody adds a comment afterwards and implicitly changes the page by this action. Generating a static version will happen very seldom after the "hot phase" when getting data directly out of the database is more appropriate. So you need a periodic task which checks if there are new "dead" stories (e.g. no comments for 2 days) and marks them with a flag for static generation on demand. It should be clear by now, that an integrated Webware scheduling mechnism is very useful for this kind of things and the better approach than external `cron` jobs. Let's look a little bit closer at the static generation technique now. First of all we need a `PageGenerator` class. To keep the example simple we just write the actual date into a file. In real life you will assemble much more complex data into such static pages.

```
from time import asctime
from TaskKit.Task import Task

html = '''<html>
<head><title>%s</title></head>
<body bgcolor="white">
<h1>%s</h1>
</body>
</html>
'''

class PageGenerator(Task):

    def __init__(self, filename):
        Task.__init__(self)
        self._filename = filename

    def run(self):
        f = open(self._filename, 'w')
        f.write(html % ('Static Page',  asctime()))
        f.close()
```

15.3.1 Scheduling

That was easy. Now it's time to schedule our task. In the following example you can see how this is accomplished with TaskKit. As a general recommendation you should put all your tasks in a separate folder (with an empty `__init__.py` file to make this folder a Python package). First of all we create a new `Scheduler` object, start it as a thread and add a periodic page generation object (of type `PageGenerator`) with the `addPeriodicAction` method. The first parameter here is the first execution time (which can be in the future), the second is the period (in seconds), the third an instance of our task class and the last parameter is a unique task name which allows us to find the task later on (e.g. if we want to change the period or put the task on hold).

```
from time import sleep, time
from TaskKit.Scheduler import Scheduler
from Tasks.PageGenerator import PageGenerator

def main():
```

(continues on next page)

(continued from previous page)

```
    scheduler = Scheduler()
    scheduler.start()
    scheduler.addPeriodicAction(time(), 5, PageGenerator('static.html'), 'PageGenerator')
    sleep(20)
    scheduler.stop()

if __name__ == '__main__':
    main()
```

When you fire up this example you will notice that the timing is not 100% accurate. The reason for this seems to be an imprecise `wait()` function in the Python `threading` module. Unfortunately this method is indispensible because we need to be able to wake up a sleeping scheduler when scheduling new tasks with first execution times smaller than `scheduler.nextTime()`. This is achieved through the `notify()` method, which sets the `notifyEvent` (`scheduler._notifyEvent.set()`). On Unix we could use `sleep` and a `signal` to interrupt this system call, but TaskKit has to be plattform independent to be of any use. But don't worry; this impreciseness is not important for normal usage, because we are talking about scheduling in the minute (not second) range here. Unix `cron` jobs have a granularity of one minute, which is a good value for TaskKit too. Of course nobody can stop you starting tasks with a period of one second (but you have been warned that this is not a good idea, except for testing purposes).

15.4 Generating static pages again

Let's refine our example a little bit and plug it into Webware. We will write a Python servlet which loks like this:

When you click on the `Generate` button a new periodic `PageGenerator` task will be added to the Webware scheduler. Remember that this will generate a static page `static.html` every 60 seconds (if you use the default values). The new task name is `"PageGenerator for filename"`, so you can use this servlet to change the settings of already scheduled tasks (by rescheduling) or add new `PageGenerator` tasks with different filenames. This is quite useless here, but as soon as you begin to parametrize your `Task` classes this approach can become quite powerful (consider for example a mail reminder form or collecting news from different news channels as periodic tasks with user defined parameters). In any case, don't be shy and contribute other interesting examples (the sky's the limit!).

Finally we come to the servlet code, which should be more or less self explanatory, except for the `_action_` construct which is very well explained in the Webware documentation though (just in case you forgot that). `app.taskManager()` gives you the Webware scheduler, which can be used to add new tasks. In real life you will have to make the scheduling information persistent and reschedule all tasks after a Webware server restart because it would be quite annoying to enter this data again and again.

```
from time import time
from ExamplePage import ExamplePage
from Tasks.PageGenerator import PageGenerator

class Schedule(ExamplePage):

    def writeContent(self):
        self.write('''
            <center><form method="post">
            <input type="submit" name="_action_ value=Generate">
            <input type="text" name="filename" value="static.html" size="16"> every
            <input type="text" name="seconds" value="60" size="4"> seconds
            </form>
            <table style="width:28em;margin-top:6px">
```

(continues on next page)

(continued from previous page)

```
            <tr style="background-color:009">
            <th colspan="2" style="color:#fff">Task List</th></tr>
            <tr style="background-color:#ddd">
            <td><b>Task Name</b></td>
            <td><b>Period</b></td></tr>''')
        for taskname, handler in self.application().taskManager().scheduledTasks().
↪items():
            self.write('''
                <tr><td>%s</td><td>%s</td></tr>''' % (taskname, handler.period()))
        self.write('''
            </table></center>''')

    def generate(self, trans):
        app = self.application()
        tm = app.taskManager()
        req = self.request()
        if req.hasField('filename') and req.hasField('seconds'):
            self._filename = req.field('filename')
            self._seconds = int(req.field('seconds'))
            task = PageGenerator(app.serverSidePath('Examples/' + self._filename))
            taskname = 'PageGenerator for ' + self._filename
            tm.addPeriodicAction(time(), self._seconds, task, taskname)
        self.writeBody()

    def methodNameForAction(self, name):
        return name.lower()

    def actions(self):
        return ExamplePage.actions(self) + ['generate']
```

15.5 The Scheduler

Now it's time to take a closer look at the `Scheduler` class itself. As you have seen in the examples above, writing tasks is only a matter of overloading the `run()` method in a derived class and adding it to the scheduler with `addTimedAction`, `addActionOnDemand`, `addDailyAction` or `addPeriodicAction`. The scheduler will wrap the Task in a `TaskHandler` structure which knows all the scheduling details and add it to its `_scheduled` or `_onDemand` dictionaries. The latter is populated by `addActionOnDemand` and contains tasks which can be called any time by `scheduler.runTaskNow('taskname')` as you can see in the following example. After that the task has gone.

```
scheduler = Scheduler()
scheduler.start()
scheduler.addActionOnDemand(SimpleTask(), 'SimpleTask')
sleep(5)
print "Demanding SimpleTask"
scheduler.runTaskNow('SimpleTask')
sleep(5)
scheduler.stop()
```

If you need a task more than one time it's better to start it regularly with one of the `add*Action` methods first. It will be added to the `_scheduled` dictionary. If you do not need the task for a certain time disable it with `scheduler.disableTask('taskname')` and enable it later with `scheduler.enableTask('taskname')`. There are some more methods (e.g. `demandTask(), stopTask(), ...`) in the `Scheduler` class which are all documented by docstrings. Take a look at them and write your own examples to understand the methods.

When a periodic task is scheduled it is added in a wrapped version to the `_scheduled` dictionary first. The (most of the time sleeping) scheduler thread always knows when to wake up and start the next task whose wrapper is moved to the `_runnning` dictionary. After completion of the task thread the handler reschedules the task (by putting it back from `_running` to `_scheduled`), calculating the next execution time `nextTime` and possibly waking up the scheduler. It is important to know that you can manipulate the handle while the task is running, e.g. change the period or call `runOnCompletion` to request that a task be re-run after its current completion. For normal use you will probably not need the handles at all, but the more you want to manipulate the task execution, the more you will appreciate the TaskHandler API. You get all the available handles from the scheduler with the `running('taskname')`, `scheduled('taskname')` and `onDemand('taskname')` methods.

In our last example which was contributed by Jay Love, who debugged, stress tested and contributed a lot of refinements to TaskKit, you see how to write a period modifying Task. This is quite weird but shows the power of handle manipulations. The last thing to remember is that the scheduler does not start a separate thread for each periodic task. It uses a thread for each task run instead and at any time keeps the number of threads as small as possible.

```
class SimpleTask(Task):

    def run(self):
        if self.proceed():
            print self.name(), time()
            print "Increasing period"
            self.handle().setPeriod(self.handle().period() + 2)
        else:
            print "Should not proceed", self.name()
```

As you can see, the TaskKit framework is quite sophisticated and will hopefully be used by many people from the Python community. If you have further question, please feel free to ask them on the Webware mailing list.

15.6 Credit

Authors: Tom Schwaller, Jay Love

Based on code from the Ganymede Directory Management System written by Jonathan Abbey.

CHAPTER

SIXTEEN

WEBUTILS

The WebUtils package is a basic set of modules for common web related programming tasks such as encoding/decoding HTML, dealing with Cookies, etc.

See the *reference documentation* for an overview of the available functions.

16.1 HTMLForException

This module defines a function by the same name:

```
def htmlForException(excInfo=None, options=None):
    ...
```

htmlForException returns an HTML string that presents useful information to the developer about the exception. The first argument is a tuple such as returned by `sys.exc_info()` which is in fact, invoked if the tuple isn't provided. The options parameter can be a dictionary to override the color options in `HTMLForExceptionOptions` which is currently defined as:

```
HTMLForExceptionOptions = {
    'table': 'background-color:#f0f0f0',
    'default': 'color:#000',
    'row.location': 'color:#009',
    'row.code': 'color:#900',
}
```

A sample HTML exception string looks like this:

16.2 HTTPStatusCodes

This module provides a list of well known HTTP status codes in list form and in a dictionary that can be keyed by code number or identifier.

You can index the HTTPStatusCodes dictionary by code number such as `200`, or identifier such as `OK`. The dictionary returned has keys `'code'`, `'identifier'` and `'htmlMsg'`. An `'asciiMsg'` key is provided, however, the HTML tags are not yet actually stripped out.

The `htmlTableOfHTTPStatusCodes()` function returns a string which is exactly that: a table containing the `HTTPStatusCodes` defined by the module. You can affect the formatting of the table by specifying values for the arguments. It's highly recommended that you use `key=value` arguments since the number and order could easily change in future versions. The definition is:

```
def htmlTableOfHTTPStatusCodes(
        codes=HTTPStatusCodeList,
        tableArgs='', rowArgs='style="vertical-align:top"',
        colArgs='', headingArgs=''):
    ...
```

If you run the script, it will invoke `htmlTableOfHTTPStatusCodes()` and print its contents with some minimal HTML wrapping. You could do this:

```
> cd Webware/Projects/WebUtils
> python HTTPStatusCodes.py > HTTPStatusCodes.html
```

And then open the HTML file in your favorite browser.

CHAPTER SEVENTEEN

MISCUTILS

The MiscUtils package provides support classes and functions to Webware that aren't necessarily web-related and that don't fit into one of the other frameworks. There is plenty of useful reusable code here.

See the *reference documentation* for an overview of the available functions.

CHAPTER

EIGHTEEN

TESTING

In this section we want to give some advice on testing Webware applications and Webware itself.

18.1 Testing Webware itself

The unit tests and end to end tests for Webware for Python can be found in the `Tests` subdirectories of the root `webware` package and its plug-ins. Webware also has a built-in context `Testing` that contains some special servlets for testing various functionality of Webware, which can be invoked manually, but will also be tested automatically as part of the end-to-end tests.

Before running the test suite, install Webware for Python into a virtual environment and activate that environment. While developing and testing Webware, it is recommended to install Webware in editable mode. To do this, unpack the source installation package of Webware for Python 3, and run this command in the directory containing the `setup.py` file:

```
pip install -e .[tests]
```

Next, change into the directory containing the main Webware package:

```
cd webware
```

To test everything, run:

```
python -m unittest discover -p Test*.py
```

To test everything, and stop on the first failing test:

```
python -m unittest discover -p Test*.py -f
```

To test everything, and print verbose output:

```
python -m unittest discover -p Test*.py -v
```

To test only UserKit:

```
python -m unittest discover -p Test*.py -vs UserKit
```

To test only the example servlets in the default context:

```
python -m unittest discover -p TestExamples.py -vs Tests.TestEndToEnd
```

You can also use tox as a test runner. The Webware source package already contains a suitable tox.ini configuration file for running the unit tests with all supported Python versions, and also running a few additional code quality checks. Make sure to use current versions of _tox and _virtualenv when running the tests.

18.2 Testing Webware applications

We recommend writing tests for your Webware applications as well, using either Python's built-in unittest framework, or the excellent pytest testing tool.

You should create unit tests for your supporting code (your own library packages in your application working directory), and also end-to-end tests for the servlets that make up your web application (the contexts in your application working directory).

For writing end-to-end tests we recommend using the WebTest package. This allows testing your Webware applications end-to-end without the overhead of starting an HTTP server, by making use of the fact that Webware applications are WSGI compliant. Have a look at the existing tests for the built-in contexts in the Tests/TestEndToEnd directory of Webware for Python 3 in order to understand how you can make use of WebTest and structure your tests.

CHAPTER

NINETEEN

API REFERENCE

19.1 Core Classes

19.1.1 Application

The Application singleton.

Application is the main class that sets up and dispatches requests. This is done using the WSGI protocol, so an *AppServer* class is not needed and not contained in Webware for Python anymore. *Application* receives the input via WSGI and turns it into *Transaction*, *HTTPRequest*, *HTTPResponse*, and *Session*.

Settings for Application are taken from `Configs/Application.config`, which is used for many global settings, even if they aren't closely tied to the Application object itself.

`class Application.`**`Application`**(*path=None*, *settings=None*, *development=None*)

Bases: *`ConfigurableForServerSidePath`*

The Application singleton.

Purpose and usage are explained in the module docstring.

`__init__`(*path=None*, *settings=None*, *development=None*)

Sets up the Application.

You can specify the path of the application working directory, a dictionary of settings to override in the configuration, and whether the application should run in development mode.

In the setting 'ApplicationConfigFilename' you can also specify a different location of the application configuration file.

`addContext`(*name*, *path*)

Add a context by named *name*, rooted at *path*.

This gets imported as a package, and the last directory of *path* does not have to match the context name. (The package will be named *name*, regardless of *path*).

Delegated to *URLParser.ContextParser*.

`static` **`addServletFactory`**(*factory*)

Add a ServletFactory.

Delegated to the *URLParser.ServletFactoryManager* singleton.

`addShutDownHandler`(*func*)

Add a shutdown handler.

Functions added through *addShutDownHandler* will be called when the Application is shutting down. You can use this hook to close database connections, clean up resources, save data to disk, etc.

`callMethodOfServlet`(*trans*, *url*, *method*, **args*, ***kw*)

Call method of another servlet.

Call a method of the servlet referred to by the URL. Calls sleep() and awake() before and after the method call. Or, if the servlet defines it, then *runMethodForTransaction* is used (analogous to the use of *runTransaction* in *forward*).

The entire process is similar to *forward*, except that instead of *respond*, *method* is called (*method* should be a string, **args* and ***kw* are passed as arguments to that method).

`commandLineConfig`()

Return the settings that came from the command-line.

These settings come via addCommandLineSetting().

`config`()

Return the configuration of the object as a dictionary.

This is a combination of defaultConfig() and userConfig(). This method caches the config.

`configFilename`()

The configuration file path.

`configName`()

Return the name of the configuration file without the extension.

This is the portion of the config file name before the '.config'. This is used on the command-line.

`configReplacementValues`()

Get config values that need to be escaped.

`contexts`()

Return a dictionary of context-name: context-path.

`static createRequestForDict`(*requestDict*)

Create request object for a given dictionary.

Create a request object (subclass of *Request*) given the raw dictionary as passed by the web server via WSGI.

The class of the request may be based on the contents of the dictionary (though only *HTTPRequest* is currently created), and the request will later determine the class of the response.

Called by *dispatchRawRequest*.

`createSessionForTransaction`(*trans*)

Get the session object for the transaction.

If the session already exists, returns that, otherwise creates a new session.

Finding the session ID is done in *Transaction.sessionId*.

`createSessionWithID`(*trans*, *sessionID*)

Create a session object with our session ID.

`defaultConfig()`

The default Application.config.

`development()`

Whether the application shall run in development mode

`dispatchRawRequest(`*requestDict*, *strmOut*`)`

Dispatch a raw request.

Dispatch a request as passed from the web server via WSGI.

This method creates the request, response, and transaction objects, then runs (via *runTransaction*) the transaction. It also catches any exceptions, which are then passed on to *handleExceptionInTransaction*.

`errorPage(`*errorClass*`)`

Get the error page url corresponding to an error class.

`forward(`*trans*, *url*`)`

Forward the request to a different (internal) URL.

The transaction's URL is changed to point to the new servlet, and the transaction is simply run again.

Output is _not_ accumulated, so if the original servlet had any output, the new output will _replace_ the old output.

You can change the request in place to control the servlet you are forwarding to – using methods like *HTTPRequest.setField*.

`handleException()`

Handle exceptions.

This should only be used in cases where there is no transaction object, for example if an exception occurs when attempting to save a session to disk.

`handleExceptionInTransaction(`*excInfo*, *trans*`)`

Handle exception with info.

Handles exception *excInfo* (as returned by *sys.exc_info()*) that was generated by *transaction*. It may display the exception report, email the report, etc., handled by *ExceptionHandler.ExceptionHandler*.

`handleMissingPathSession(`*trans*`)`

Redirect requests without session info in the path.

If UseAutomaticPathSessions is enabled in Application.config we redirect the browser to an absolute url with SID in path http://gandalf/a/_SID_=2001080221301877755/Examples/ _SID_ is extracted and removed from path in HTTPRequest.py

This is for convenient building of webapps that must not depend on cookie support.

Note that we create an absolute URL with scheme and hostname because otherwise IIS will only cause an internal redirect.

`handlePathSession(`*trans*`)`

Handle the session identifier that has been found in the path.

`handleUnnecessaryPathSession(`*trans*`)`

Redirect request with unnecessary session info in the path.

This is called if it has been determined that the request has a path session, but also cookies. In that case we redirect to eliminate the unnecessary path session.

`hasContext`(*name*)

Checks whether context *name* exist.

`hasSession`(*sessionId*)

Check whether session *sessionId* exists.

`hasSetting`(*name*)

Check whether a configuration setting has been changed.

`includeURL`(*trans*, *url*)

Include another servlet.

Include the servlet given by the URL. Like *forward*, except control is ultimately returned to the servlet.

`initErrorPage`()

Initialize the error page related attributes.

`initSessions`()

Initialize all session related attributes.

`initVersions`()

Get and store versions.

Initialize attributes that stores the Webware version as both tuple and string. These are stored in the Properties.py files.

`loadPlugIn`(*name*, *module*)

Load and return the given plug-in.

May return None if loading was unsuccessful (in which case this method prints a message saying so). Used by *loadPlugIns* (note the **s**).

`loadPlugIns`()

Load all plug-ins.

A plug-in allows you to extend the functionality of Webware without necessarily having to modify its source. Plug-ins are loaded by Application at startup time, just before listening for requests. See the docs in *PlugIn* for more info.

`makeDirs`()

Make sure some standard directories are always available.

`static name`()

The name by which this was started. Usually *Application*.

`numRequests`()

Return the number of requests.

Returns the number of requests received by this application since it was launched.

`outputEncoding`()

Get the default output encoding of this application.

`plugIn`(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return the plug-in with the given name.

`plugIns`()

Return a dictionary of the plug-ins loaded by the application.

Each plug-in is a PlugIn object with an underlying Python package.

`printConfig`(*dest=None*)

Print the configuration to the given destination.

The default destination is stdout. A fixed with font is assumed for aligning the values to start at the same column.

`printStartUpMessage`()

Print a little intro to the activity log.

`static readConfig`(*filename*)

Read the configuration from the file with the given name.

Raises an UIError if the configuration cannot be read.

This implementation assumes the file is stored in utf-8 encoding with possible BOM at the start, but also tries to read as latin-1 if it cannot be decoded as utf-8. Subclasses can override this behavior.

`registerShutDownHandler`()

Register shutdown handler in various ways.

We want to make sure the shutdown handler is called, so that the application can save the sessions to disk and do cleanup tasks.

`static removePathSession`(*trans*)

Remove a possible session identifier from the path.

`static resolveInternalRelativePath`(*trans*, *url*)

Return the absolute internal path.

Given a URL, return the absolute internal URL. URLs are assumed relative to the current URL. Absolute paths are returned unchanged.

`static returnServlet`(*servlet*)

Return the servlet to its pool.

`rootURLParser`()

Accessor: the Root URL parser.

URL parsing (as defined by subclasses of *URLParser.URLParser*) starts here. Other parsers are called in turn by this parser.

`runTransaction`(*trans*)

Run transaction.

Executes the transaction, handling HTTPException errors. Finds the servlet (using the root parser, probably *URLParser.ContextParser*, to find the servlet for the transaction, then calling *runTransactionViaServlet*.

Called by *dispatchRawRequest*.

`static runTransactionViaServlet`(*servlet*, *trans*)

Execute the transaction using the servlet.

This is the *awake*/*respond*/*sleep* sequence of calls, or if the servlet supports it, a single *runTransaction* call (which is presumed to make the awake/respond/sleep calls on its own). Using *runTransaction* allows the servlet to override the basic call sequence, or catch errors from that sequence.

Called by *runTransaction*.

`serverSidePath`(*path=None*)

Get the server-side path.

Returns the absolute server-side path of the Webware application. If the optional path is passed in, then it is joined with the server side directory to form a path relative to the working directory.

`session`(*sessionId*, *default=<class 'MiscUtils.NoDefault'>*)

The session object for *sessionId*.

Raises *KeyError* if session not found and no default is given.

`sessionCookiePath`(*trans*)

Get the cookie path for this transaction.

If not path is specified in the configuration setting, the servlet path is used for security reasons, see: https://www.helpnetsecurity.com/2004/06/27/cookie-path-best-practice/

`sessionName`(*_trans*)

Get the name of the field holding the session ID.

Overwrite to make this transaction dependent.

`sessionPrefix`(*_trans*)

Get the prefix string for the session ID.

Overwrite to make this transaction dependent.

`sessionTimeout`(*_trans*)

Get the timeout (in seconds) for a user session.

Overwrite to make this transaction dependent.

`sessions`()

A dictionary of all the session objects.

`setSetting`(*name*, *value*)

Set a particular configuration setting.

`setting`(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return setting, using the server side path when indicated.

Returns the setting, filtered by *self.serverSidePath()*, if the name ends with `Filename` or `Dir`.

`shutDown`()

Shut down the application.

Called when the interpreter is terminated.

`sigTerm`(*_signum*, *_frame*)

Signal handler for terminating the process.

`startSessionSweeper`()

Start session sweeper.

Starts the session sweeper, *Tasks.SessionTask*, which deletes session objects (and disk copies of those objects) that have expired.

`startTime`()

Return the time the application was started.

The time is given as seconds, like time().

taskManager()

Accessor: *TaskKit.Scheduler* instance.

userConfig()

Return the user config overrides.

These settings can be found in the optional config file. Returns {} if there is no such file.

The config filename is taken from configFilename().

webwarePath()

Return the Webware path.

webwareVersion()

Return the Webware version as a tuple.

webwareVersionString()

Return the Webware version as a printable string.

writeActivityLog(*trans*)

Write an entry to the activity log.

Writes an entry to the script log file. Uses settings `ActivityLogFilename` and `ActivityLogColumns`.

writeExceptionReport(*handler*)

Write extra information to the exception report.

See *ExceptionHandler* for more information.

exception Application.**EndResponse**

Bases: `Exception`

End response exception.

Used to prematurely break out of the *awake()/respond()/sleep()* cycle without reporting a traceback. During servlet processing, if this exception is caught during *awake()* or *respond()* then *sleep()* is called and the response is sent. If caught during *sleep()*, processing ends and the response is sent.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

19.1.2 ConfigurableForServerSidePath

class ConfigurableForServerSidePath.**ConfigurableForServerSidePath**

Bases: `Configurable`

Configuration file functionality incorporating a server side path.

This is a version of *MiscUtils.Configurable.Configurable* that provides a customized *setting* method for classes which have a *serverSidePath* method. If a setting's name ends with `Filename` or `Dir`, its value is passed through *serverSidePath* before being returned.

In other words, relative filenames and directory names are expanded with the location of the object, *not* the current directory.

Application is a prominent class that uses this mix-in. Any class that has a *serverSidePath* method and a *Configurable* base class, should inherit this class instead.

This is used for *MakeAppWorkDir*, which changes the *serverSidePath*.

`__init__()`

`commandLineConfig()`

Return the settings that came from the command-line.

These settings come via addCommandLineSetting().

`config()`

Return the configuration of the object as a dictionary.

This is a combination of defaultConfig() and userConfig(). This method caches the config.

`configFilename()`

Return the full name of the user config file.

Users can override the configuration by this config file. Subclasses must override to specify a name. Returning None is valid, in which case no user config file will be loaded.

`configName()`

Return the name of the configuration file without the extension.

This is the portion of the config file name before the '.config'. This is used on the command-line.

`configReplacementValues()`

Return a dictionary for substitutions in the config file.

This must be a dictionary suitable for use with "string % dict" that should be used on the text in the config file. If an empty dictionary (or None) is returned, then no substitution will be attempted.

`defaultConfig()`

Return a dictionary with all the default values for the settings.

This implementation returns {}. Subclasses should override.

`hasSetting(`*name*`)`

Check whether a configuration setting has been changed.

`printConfig(`*dest=None*`)`

Print the configuration to the given destination.

The default destination is stdout. A fixed with font is assumed for aligning the values to start at the same column.

`static readConfig(`*filename*`)`

Read the configuration from the file with the given name.

Raises an UIError if the configuration cannot be read.

This implementation assumes the file is stored in utf-8 encoding with possible BOM at the start, but also tries to read as latin-1 if it cannot be decoded as utf-8. Subclasses can override this behavior.

`setSetting(`*name*, *value*`)`

Set a particular configuration setting.

setting(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return setting, using the server side path when indicated.

Returns the setting, filtered by *self.serverSidePath()*, if the name ends with `Filename` or `Dir`.

userConfig()

Return the user config overrides.

These settings can be found in the optional config file. Returns {} if there is no such file.

The config filename is taken from configFilename().

19.1.3 Cookie

class Cookie.**Cookie**(*name*, *value*)

Bases: `object`

Delicious cookies.

Cookie is used to create cookies that have additional attributes beyond their value.

Note that web browsers don't typically send any information with the cookie other than its value. Therefore *HTTPRequest.cookie* simply returns a value such as an integer or a string.

When the server sends cookies back to the browser, it can send a cookie that simply has a value, or the cookie can be accompanied by various attributes (domain, path, max-age, . . .) as described in RFC 2109. Therefore, in *HTTPResponse*, *setCookie* can take either an instance of the *Cookie* class, as defined in this module, or a value.

Note that *Cookie* values get pickled (see the *pickle* module), so you can set and get cookies that are integers, lists, dictionaries, etc.

__init__(*name*, *value*)

Create a cookie.

Properties other than *name* and *value* are set with methods.

comment()

delete()

Delete a cookie.

When sent, this should delete the cookie from the user's browser, by making it empty, expiring it in the past, and setting its max-age to 0. One of these will delete the cookie for any browser (which one actually works depends on the browser).

domain()

expires()

headerValue()

Return header value.

Returns a string with the value that should be used in the HTTP headers.

httpOnly()

isSecure()

maxAge()

name()

path()

sameSite()

setComment(*comment*)

setDomain(*domain*)

setExpires(*expires*)

setHttpOnly(*httpOnly=True*)

setMaxAge(*maxAge*)

setPath(*path*)

setSameSite(*sameSite='Strict'*)

setSecure(*secure=True*)

setValue(*value*)

setVersion(*version*)

value()

version()

19.1.4 ExceptionHandler

Exception handling.

class ExceptionHandler.**ExceptionHandler**(*application*, *transaction*, *excInfo*, *formatOptions=None*)

Bases: `object`

Exception handling.

ExceptionHandler is a utility class for Application that is created to handle a particular exception. The object is a one-shot deal. After handling an exception, it should be removed.

At some point, the exception handler sends *writeExceptionReport* to the transaction (if present), which in turn sends it to the other transactional objects (application, request, response, etc.) The handler is the single argument for this message.

Classes may find it useful to do things like this:

```python
exceptionReportAttrs = ['foo', 'bar', 'baz']
def writeExceptionReport(self, handler):
    handler.writeTitle(self.__class__.__name__)
    handler.writeAttrs(self, self.exceptionReportAttrs)
```

The handler write methods that may be useful are:

- write
- writeTitle
- writeDict

- writeAttrs

Derived classes must not assume that the error occurred in a transaction. self._tra may be None for exceptions outside of transactions.

HOW TO CREATE A CUSTOM EXCEPTION HANDLER

In the `__init__.py` of your context:

```
from ExceptionHandler import ExceptionHandler as _ExceptionHandler

class ExceptionHandler(_ExceptionHandler):

    _hideValuesForFields = _ExceptionHandler._hideValuesForFields + [
        'foo', 'bar']

    def work(self):
        _ExceptionHandler.work(self)
        # do whatever
        # override other methods if you like

def contextInitialize(app, ctxPath):
    app._exceptionHandlerClass = ExceptionHandler
```

You can also control the errors with settings in `Application.config`.

`__init__`(*application*, *transaction*, *excInfo*, *formatOptions=None*)

Create an exception handler instance.

ExceptionHandler instances are created anew for each exception. Instantiating ExceptionHandler completes the process – the caller need not do anything else.

`basicServletName`()

The base name for the servlet (sans directory).

`emailException`(*htmlErrMsg*)

Email the exception.

Send the exception via mail, either as an attachment, or as the body of the mail.

`errorPageFilename`()

Create filename for error page.

Construct a filename for an HTML error page, not including the `ErrorMessagesDir` setting (which *saveError* adds on).

`filterDictValue`(*value*, *key*, *_dict*)

Filter dictionary values.

Filters keys from a dict. Currently ignores the dictionary, and just filters based on the key.

`filterValue`(*value*, *key*)

Filter values.

This is the core filter method that is used in all filtering. By default, it simply returns self._hiddenString if the key is in self._hideValuesForField (case insensitive). Subclasses could override for more elaborate filtering techniques.

htmlDebugInfo()

Return the debug info.

Return HTML-formatted debugging information on the current exception. Calls *writeHTML*, which uses `self.write(...)` to add content.

logExceptionToConsole(*stderr=None*)

Log an exception.

Logs the time, servlet name and traceback to the console (typically stderr). This usually results in the information appearing in console/terminal from which the Application was launched.

logExceptionToDisk(*errorMsgFilename=None*)

Log the exception to disk.

Writes a tuple containing (date-time, filename, pathname, exception-name, exception-data, error report filename) to the errors file (typically 'Errors.csv') in CSV format. Invoked by *handleException*.

privateErrorPage()

Return a private error page.

Returns an HTML page intended for the developer with useful information such as the traceback.

Most of the contents are generated in *htmlDebugInfo*.

publicErrorPage()

Return a public error page.

Returns a brief error page telling the user that an error has occurred. Body of the message comes from `UserErrorMessage` setting.

repr(*value*)

Get HTML encoded representation.

Returns the repr() of value already HTML encoded. As a special case, dictionaries are nicely formatted in table.

This is a utility method for *writeAttrs*.

saveErrorPage(*html*)

Save the error page.

Saves the given HTML error page for later viewing by the developer, and returns the filename used.

servletPathname()

The full filesystem path for the servlet.

setting(*name*)

Settings are inherited from Application.

work()

Main error handling method.

Invoked by `__init__` to do the main work. This calls *logExceptionToConsole*, then checks settings to see if it should call *saveErrorPage* (to save the error to disk) and *emailException*.

It also sends gives a page from *privateErrorPage* or *publicErrorPage* (which one based on *ShowDebugInfoOnErrors*).

write(*s*)

Output *s* to the body.

writeAttrs(*obj*, *attrNames*)

Output object attributes.

Writes the attributes of the object as given by attrNames. Tries `obj._name` first, followed by `obj.name()`. Is resilient regarding exceptions so as not to spoil the exception report.

writeDict(*d*, *heading=None*, *encoded=None*)

Output a table-formatted dictionary.

writeEnvironment()

Output environment.

Writes the environment this is being run in. This is *not* the environment that was passed in with the request (holding the CGI information) – it's just the information from the environment that the Application is being executed in.

writeFancyTraceback()

Output a fancy traceback, using CGITraceback.

writeHTML()

Write the traceback.

Writes all the parts of the traceback, invoking:

- *writeTraceback*
- *writeMiscInfo*
- *writeTransaction*
- *writeEnvironment*
- *writeIds*
- *writeFancyTraceback*

writeIds()

Output OS identification.

Prints some values from the OS (like processor ID).

writeMiscInfo()

Output misc info.

Write a couple little pieces of information about the environment.

writeTitle(*s*)

Output the sub-heading to define a section.

writeTraceback()

Output the traceback.

Writes the traceback, with most of the work done by *WebUtils.HTMLForException.htmlForException*.

writeTransaction()

Output transaction.

Lets the transaction talk about itself, using *Transaction.writeExceptionReport*.

writeln(*s*)

Output *s* plus a newline.

class ExceptionHandler.**Singleton**

Bases: `object`

A singleton object.

ExceptionHandler.**docType**()

Return the document type for the page.

ExceptionHandler.**htStyle**()

Return the page style.

ExceptionHandler.**htTitle**(*name*)

Format a *name* as a section title.

ExceptionHandler.**osIdDict**()

Get all OS id information.

Returns a dictionary containing id information such as pid and uid.

19.1.5 HTTPContent

Content producing HTTP servlet.

class HTTPContent.**HTTPContent**

Bases: *HTTPServlet*

Content producing HTTP servlet.

HTTPContent is a type of HTTPServlet that is more convenient for Servlets which represent content generated in response to GET and POST requests. If you are generating HTML content, you you probably want your servlet to inherit from Page, which contains many HTML-related convenience methods.

If you are generating non-HTML content, it is appropriate to inherit from this class directly.

Subclasses typically override defaultAction().

In *awake*, the page sets self attributes: *_transaction*, *_response* and *_request* which subclasses should use as appropriate.

For the purposes of output, the *write* and *writeln* convenience methods are provided.

If you plan to produce HTML content, you should start by looking at Page instead of this lower-level class.

__init__()

Subclasses must invoke super.

actions()

The allowed actions.

Returns a list or a set of method names that are allowable actions from HTML forms. The default implementation returns []. See *_respond* for more about actions.

application()

The *Application* instance we're using.

awake(*transaction*)

Let servlet awake.

Makes instance variables from the transaction. This is where Page becomes unthreadsafe, as the page is tied to the transaction. This is also what allows us to implement functions like *write*, where you don't need to pass in the transaction or response.

`callMethodOfServlet`(*url*, *method*, **args*, ***kwargs*)

Call a method of another servlet.

See *Application.callMethodOfServlet* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`canBeReused`()

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

`canBeThreaded`()

Declares whether servlet can be threaded.

Returns False because of the instance variables we set up in *awake*.

`close`()

`defaultAction`()

Default action.

The core method that gets called as a result of requests. Subclasses should override this.

`static endResponse`()

End response.

When this method is called during *awake* or *respond*, servlet processing will end immediately, and the accumulated response will be sent.

Note that *sleep* will still be called, providing a chance to clean up or free any resources.

`forward`(*url*)

Forward request.

Forwards this request to another servlet. See *Application.forward* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`handleAction`(*action*)

Handle action.

Invoked by *_respond* when a legitimate action has been found in a form. Invokes *preAction*, the actual action method and *postAction*.

Subclasses rarely override this method.

`includeURL`(*url*)

Include output from other servlet.

Includes the response of another servlet in the current servlet's response. See *Application.includeURL* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`lastModified`(*_trans*)

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`log`(*message*)

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`methodNameForAction`(*name*)

Return method name for an action name.

Invoked by _respond() to determine the method name for a given action name which has been derived as the value of an `_action_` field. Since this is usually the label of an HTML submit button in a form, it is often needed to transform it in order to get a valid method name (for instance, blanks could be replaced by underscores and the like). This default implementation of the name transformation is the identity, it simply returns the name. Subclasses should override this method when action names don't match their method names; they could "mangle" the action names or look the method names up in a dictionary.

`name`()

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented`(*trans*)

`open`()

`outputEncoding`()

Get the default output encoding of the application.

`postAction`(*actionName*)

Things to do after action.

Invoked by self after invoking an action method. Subclasses may override to customize and may or may not invoke super as they see fit. The *actionName* is passed to this method, although it seems a generally bad idea to rely on this. However, it's still provided just in case you need that hook.

By default, this does nothing.

`preAction`(*actionName*)

Things to do before action.

Invoked by self prior to invoking an action method. The *actionName* is passed to this method, although it seems a generally bad idea to rely on this. However, it's still provided just in case you need that hook.

By default, this does nothing.

`request`()

The request (*HTTPRequest*) we're handling.

`respond`(*transaction*)

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, …).

`respondToGet`(*transaction*)

Respond to GET.

Invoked in response to a GET request method. All methods are passed to *_respond*.

respondToHead(*trans*)

Respond to a HEAD request.

A correct but inefficient implementation.

respondToPost(*transaction*)

Respond to POST.

Invoked in response to a POST request method. All methods are passed to *_respond*.

response()

The response (*HTTPResponse*) we're handling.

runMethodForTransaction(*transaction*, *method*, **args*, ***kw*)

static **runTransaction**(*transaction*)

sendRedirectAndEnd(*url*, *status=None*)

Send redirect and end.

Sends a redirect back to the client and ends the response. This is a very popular pattern.

sendRedirectPermanentAndEnd(*url*)

Send permanent redirect and end.

sendRedirectSeeOtherAndEnd(*url*)

Send redirect to a URL to be retrieved with GET and end.

This is the proper method for the Post/Redirect/Get pattern.

sendRedirectTemporaryAndEnd(*url*)

Send temporary redirect and end.

serverSidePath(*path=None*)

Return the filesystem path of the page on the server.

session()

The session object.

This provides a state for the current user (associated with a browser instance, really). If no session exists, then a session will be created.

sessionEncode(*url=None*)

Utility function to access *Session.sessionEncode*.

Takes a url and adds the session ID as a parameter. This is for cases where you don't know if the client will accepts cookies.

setFactory(*factory*)

sleep(*transaction*)

Let servlet sleep again.

We unset some variables. Very boring.

transaction()

The *Transaction* we're currently handling.

static **urlDecode**(*s*)

Turn special % characters into actual characters.

This method does the same as the *urllib.unquote_plus()* function.

static urlEncode(*s*)

Quotes special characters using the % substitutions.

This method does the same as the *urllib.quote_plus()* function.

write(**args*)

Write to output.

Writes the arguments, which are turned to strings (with *str*) and concatenated before being written to the response. Unicode strings must be encoded before they can be written.

writeExceptionReport(*handler*)

Write extra information to the exception report.

The *handler* argument is the exception handler, and information is written there (using *writeTitle*, *write*, and *writeln*). This information is added to the exception report.

See *ExceptionHandler* for more information.

writeln(**args*)

Write to output with newline.

Writes the arguments (like *write*), adding a newline after. Unicode strings must be encoded before they can be written.

exception `HTTPContent.`**HTTPContentError**

Bases: `Exception`

HTTP content error

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

19.1.6 HTTPExceptions

HTTP exceptions.

HTTPExceptions are for situations that are predicted by the HTTP spec. Where the `200 OK` response is typical, a `404 Not Found` or `301 Moved Temporarily` response is not entirely unexpected.

Application catches all *HTTPException* exceptions (and subclasses of HTTPException), and instead of being errors these are translated into responses. In various places these can also be caught and changed, for instance an *HTTPAuthenticationRequired* could be turned into a normal login page.

exception `HTTPExceptions.`**HTTPAuthenticationRequired**(*realm=None*)

Bases: *HTTPException*

HTTPException “authentication required” subclass.

HTTPAuthenticationRequired will usually cause the browser to open up an HTTP login box, and after getting login information from the user, the browser will resubmit the request. However, this should also trigger login pages in properly set up environments (though much code will not work this way).

Browsers will usually not send authentication information unless they receive this response, even when other pages on the site have given 401 responses before. So when using this authentication every request will usually be doubled, once without authentication, once with.

`__init__`(*realm=None*)

`args`

`code`()

The integer code.

`codeMessage`()

The message (like `Not Found`) that goes with the code.

`description`()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

`headers`()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

`htBody`()

The HTML body of the page.

`htDescription`()

HTML error description.

The HTML description of the error, for presentation to the browser user.

`htTitle`()

The title, but it may include HTML markup (like italics).

`html`()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

`setTransaction`(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

`title`()

The title used in the HTML page.

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

`HTTPExceptions.`**`HTTPAuthorizationRequired`**

alias of *`HTTPAuthenticationRequired`*

exception `HTTPExceptions.`**`HTTPBadRequest`**

Bases: *`HTTPException`*

HTTPException "bad request" subclass.

When the browser sends an invalid request.

`__init__`(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPConflict**

Bases: *HTTPException*

HTTPException “conflict” subclass.

When there’s a locking conflict on this resource (in response to something like a PUT, not for most other conflicts). Mostly for WebDAV.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPException**

Bases: `Exception`

HTTPException template class.

Subclasses must define these variables (usually as class variables):

***_code*:**

a tuple of the integer error code, and the short description that goes with it (like `(200, "OK")`)

***_description*:**

the long-winded description, to be presented in the response page. Or you can override description() if you want something more context-sensitive.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPForbidden**

Bases: *HTTPException*

HTTPException "forbidden" subclass.

When access is not allowed to this resource. If the user is anonymous, and must be authenticated, then HTTPAuthenticationRequired is a preferable exception. If the user should not be able to get to this resource (at least through the path they did), or is authenticated and still doesn't have access, or no one is allowed to view this, then HTTPForbidden would be the proper response.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPInsufficientStorage**

Bases: *HTTPException*

HTTPException “insufficient storage” subclass.

When there is not sufficient storage, usually in response to a PUT when there isn’t enough disk space. Mostly for WebDAV.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPMethodNotAllowed**

Bases: *HTTPException*

HTTPException "method not allowed" subclass.

When a method (like GET, PROPFIND, POST, etc) is not allowed on this resource (usually because it does not make sense, not because it is not permitted). Mostly for WebDAV.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPMovedPermanently**(*location=None*, *webwareLocation=None*)

Bases: *HTTPException*

HTTPException "moved permanently" subclass.

When a resource is permanently moved. The browser may remember this relocation, and later requests may skip requesting the original resource altogether.

__init__(*location=None*, *webwareLocation=None*)

Set destination.

HTTPMovedPermanently needs a destination that you it should be directed to – you can pass *location or webwareLocation* – if you pass *webwareLocation* it will be relative to the Webware root location (the mount point of the WSGI application).

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

We include a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

location()

The location that we will be redirecting to.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPNotFound**

Bases: *HTTPException*

HTTPException "not found" subclass.

When the requested resource does not exist. To be more secretive, it is okay to return a 404 if access to the resource is not permitted (you are not required to use HTTPForbidden, though it makes it more clear why access was disallowed).

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPNotImplemented**

Bases: *HTTPException*

HTTPException "not implemented" subclass.

When methods (like GET, POST, PUT, PROPFIND, etc) are not implemented for this resource.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPPreconditionFailed**

Bases: *HTTPException*

HTTPException "Precondition Failed" subclass.

During compound, atomic operations, when a precondition for an early operation fail, then later operations in will fail with this code. Mostly for WebDAV.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

HTTPExceptions.**HTTPRedirect**

alias of *HTTPTemporaryRedirect*

exception HTTPExceptions.**HTTPRequestTimeout**

Bases: *HTTPException*

HTTPException "request timeout" subclass.

The client did not produce a request within the time that the server was prepared to wait. The client may repeat the request without modifications at any later time.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPServerError**

Bases: *HTTPException*

HTTPException "Server Error" subclass.

The server encountered an unexpected condition which prevented it from fulfilling the request.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPServiceUnavailable**

Bases: *HTTPException*

HTTPException "service unavailable" subclass.

The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPSessionExpired**

Bases: *HTTPException*

HTTPException “session expired” subclass.

This is the same as HTTPAuthenticationRequired, but should be used in the situation when a session has expired.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPTemporaryRedirect**(*location=None*, *webwareLocation=None*)

Bases: *HTTPMovedPermanently*

HTTPException "temporary redirect" subclass.

Like HTTPMovedPermanently, except the redirect is only valid for this request. Internally identical to HTTP-MovedPermanently, except with a different response code. Browsers will check the server for each request to see where it's redirected to.

__init__(*location=None*, *webwareLocation=None*)

Set destination.

HTTPMovedPermanently needs a destination that you it should be directed to – you can pass *location or webwareLocation* – if you pass *webwareLocation* it will be relative to the Webware root location (the mount point of the WSGI application).

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

We include a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

location()

The location that we will be redirecting to.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception HTTPExceptions.**HTTPUnsupportedMediaType**

Bases: *HTTPException*

HTTPException "unsupported media type" subclass.

The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

__init__(**args*, ***kwargs*)

args

code()

The integer code.

codeMessage()

The message (like `Not Found`) that goes with the code.

description()

Error description.

Possibly a plain text version of the error description, though usually just identical to *htDescription*.

headers()

Get headers.

Additional headers that should be sent with the response, not including the Status header. For instance, the redirect exception adds a Location header.

htBody()

The HTML body of the page.

htDescription()

HTML error description.

The HTML description of the error, for presentation to the browser user.

htTitle()

The title, but it may include HTML markup (like italics).

html()

The error page.

The HTML page that should be sent with the error, usually a description of the problem.

setTransaction(*trans*)

Set transaction.

When the exception is caught by *Application*, it tells the exception what the transaction is. This way you can resolve relative paths, or otherwise act in a manner sensitive of the context of the error.

title()

The title used in the HTML page.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

19.1.7 HTTPRequest

HTTP requests

class HTTPRequest.**HTTPRequest**(*requestDict=None*)

Bases: `Request`

The base class for HTTP requests.

__init__(*requestDict=None*)

Initialize the request.

Subclasses are responsible for invoking super and initializing self._time.

accept(*which=None*)

Return preferences as requested by the user agent.

The accepted preferences are returned as a list of codes in the same order as they appeared in the header. In other words, the explicit weighting criteria are ignored.

If you do not define otherwise which preferences you are interested in ('language', 'charset', 'encoding'), by default you will get the user preferences for the content types.

clearTransaction()

contextName()

Return the name of the context of this request.

This isn't necessarily the same as the name of the directory containing the context.

contextPath()

Return the portion of the URI that is the context of the request.

cookie(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return the value of the specified cookie.

cookies()

Return a dict of all cookies the client sent with this request.

delField(*name*)

environ()

Get the environment for the request.

`extraURLPath()`

Return additional path components in the URL.

Only works if the Application.config setting "ExtraPathInfo" is set to true; otherwise you will get a page not found error.

`field`(*name*, *default=<class 'MiscUtils.NoDefault'>*)

`fieldStorage()`

`fields()`

`hasCookie`(*name*)

Return whether a cookie with the given name exists.

`hasField`(*name*)

`hasValue`(*name*)

Check whether there is a value with the given name.

`hostAndPort()`

Return the hostname and port part from the URL of this request.

`htmlInfo()`

Return a single HTML string that represents info().

Useful for inspecting objects via web browsers.

`info()`

Return request info.

Return a list of tuples where each tuple has a key/label (a string) and a value (any kind of object).

Values are typically atomic values such as numbers and strings or another list of tuples in the same fashion. This is for debugging only.

`input()`

Return a file-style object that the contents can be read from.

`isSecure()`

Check whether this is a HTTPS connection.

`isSessionExpired()`

Return whether the request originally had an expired session ID.

Only works if the Application.config setting "IgnoreInvalidSession" is set to true; otherwise you get a canned error page on an invalid session, so your servlet never gets processed.

`localAddress()`

Get local address.

Returns a string containing the Internet Protocol (IP) address of the local host (e.g., the server) that received the request.

`static localName()`

Get local name.

Returns the fully qualified name of the local host (e.g., the server) that received the request.

`localPort()`
Get local port.

Returns the port of the local host (e.g., the server) that received the request.

`method()`
Return the HTTP request method (in all uppercase).

Typically from the set GET, POST, PUT, DELETE, OPTIONS and TRACE.

`originalContextName()`
Return the name of the original context before any forwarding.

`originalServlet()`
Get original servlet before any forwarding.

`originalURI()`
Get URI of the original servlet before any forwarding.

`originalURLPath()`
Get URL path of the original servlet before any forwarding.

`parent()`
Get the servlet that passed this request to us, if any.

`parents()`
Get the list of all previous servlets.

`pathInfo()`
Return any extra path information as sent by the client.

This is anything after the servlet name but before the query string. Equivalent to the CGI variable PATH_INFO.

`pathTranslated()`
Return extra path information translated as file system path.

This is the same as pathInfo() but translated to the file system. Equivalent to the CGI variable PATH_TRANSLATED.

`pop()`
Pop URL path and servlet from the stack, returning the servlet.

`previousContextName()`
Get the previous context name, if any.

`previousContextNames()`
Get the list of all previous context names.

`previousServlet()`
Get the servlet that passed this request to us, if any.

`previousServlets()`
Get the list of all previous servlets.

`previousURI()`
Get the previous URI, if any.

`previousURIs()`
Get the list of all previous URIs.

`previousURLPath()`

Get the previous URL path, if any.

`previousURLPaths()`

Get the list of all previous URL paths.

`protocol()`

Return the name and version of the protocol.

`push(`*servlet*, *url=None*`)`

Push servlet and URL path on a stack, setting a new URL.

`queryString()`

Return the query string portion of the URL for this request.

Equivalent to the CGI variable QUERY_STRING.

`rawInput(`*rewind=False*`)`

Get the raw input from the request.

This gives you a file-like object for the data that was sent with the request (e.g., the body of a POST request, or the document uploaded in a PUT request).

The file might not be rewound to the beginning if there was valid, form-encoded POST data. Pass rewind=True if you want to be sure you get the entire body of the request.

`remoteAddress()`

Return a string containing the IP address of the client.

`remoteName()`

Return the fully qualified name of the client that sent the request.

Returns the IP address of the client if the name cannot be determined.

`remoteUser()`

Always returns None since authentication is not yet supported.

Take from CGI variable REMOTE_USER.

`requestID()`

Return the request ID.

The request ID is a serial number unique to this request (at least unique for one run of the Application).

`responseClass()`

Get the corresponding response class.

`scheme()`

Return the URI scheme of the request (http or https).

`scriptFileName()`

Return the filesystem path of the WSGI script.

Equivalent to the CGI variable SCRIPT_FILENAME.

`scriptName()`

Return the name of the WSGI script as it appears in the URL.

Example: '/Webware' Does not reflect redirection by the web server. Equivalent to the CGI variable SCRIPT_NAME.

`serverDictionary``()`

Return a dictionary with the data the web server gave us.

This data includes HTTP_HOST and HTTP_USER_AGENT, for example.

`serverPath``()`

Return the web server URL path of this request.

This is the URL that was actually received by the web server before any rewriting took place.

Same as serverURL, but without scheme and host.

`serverPathDir``()`

Return the directory of the web server URL path.

Same as serverPath, but removes the actual page.

`serverSideContextPath``(`*path=None*`)`

Return the absolute server-side path of the context of this request.

If the optional path is passed in, then it is joined with the server side context directory to form a path relative to the object.

This directory could be different from the result of serverSidePath() if the request is in a subdirectory of the main context directory.

`serverSidePath``(`*path=None*`)`

Return the absolute server-side path of the request.

If the optional path is passed in, then it is joined with the server side directory to form a path relative to the object.

`serverURL``(`*canonical=False*`)`

Return the full Internet path to this request.

This is the URL that was actually received by the web server before any rewriting took place. If canonical is set to true, then the canonical hostname of the server is used if possible.

The path is returned without any extra path info or query strings, i.e. https://www.my.own.host.com:8080/Webware/TestPage.py

`serverURLDir``()`

Return the directory of the URL in full Internet form.

Same as serverURL, but removes the actual page.

`servlet``()`

Get current servlet for this request.

`servletPath``()`

Return the base URL for the servlets, sans host.

This is useful in cases when you are constructing URLs. See Testing/Main.py for an example use.

Roughly equivalent to the CGI variable SCRIPT_NAME, but reflects redirection by the web server.

`servletPathFromSiteRoot``()`

Return the "servlet path" of this servlet relative to the siteRoot.

In other words, everything after the name of the context (if present). If you append this to the result of self.siteRoot() you get back to the current servlet. This is useful for saving the path to the current servlet in a database, for example.

`servletURI()`

Return servlet URI without any query strings or extra path info.

`session()`

Return the session associated with this request.

The session is either as specified by sessionId() or newly created. This is a convenience for transaction.session()

`sessionId()`

Return a string with the session ID specified by the client.

Returns None if there is no session ID.

`setField`(*name*, *value*)

`setSessionExpired`(*sessionExpired*)

`setSessionId`(*sessionID*, *force=False*)

Set the session ID.

This needs to be called _before_ attempting to use the session. This would be useful if the session ID is being passed in through unusual means, for example via a field in an XML-RPC request.

Pass in force=True if you want to force this session ID to be used even if the session doesn't exist. This would be useful in unusual circumstances where the client is responsible for creating the unique session ID rather than the server. Be sure to use only legal filename characters in the session ID – 0-9, a-z, A-Z, _, -, and . are OK but everything else will be rejected, as will identifiers longer than 80 characters. (Without passing in force=True, a random session ID will be generated if that session ID isn't already present in the session store.)

`setTransaction`(*trans*)

Set a transaction container.

`setURLPath`(*path*)

Set the URL path of the request.

There is rarely a need to do this. Proceed with caution.

`siteRoot()`

Return the relative URL path of the home location.

This includes all URL path components necessary to get back home from the current location.

Examples:
‘’ ‘../’ ‘../../’

You can use this as a prefix to a URL that you know is based off the home location. Any time you are in a servlet that may have been forwarded to from another servlet at a different level, you should prefix your URL's with this. That is, if servlet "Foo/Bar" forwards to "Qux", then the qux servlet should use siteRoot() to construct all links to avoid broken links. This works properly because this method computes the path based on the _original_ servlet, not the location of the servlet that you have forwarded to.

`siteRootFromCurrentServlet()`

Return relative URL path to home seen from the current servlet.

This includes all URL path components necessary to get back home from the current servlet (not from the original request).

Similar to siteRoot() but instead, it returns the site root relative to the _current_ servlet, not the _original_ servlet.

time()

Return the time that the request was received.

timeStamp()

Return time() as human readable string for logging and debugging.

transaction()

Get the transaction container.

uri()

Return the URI for this request (everything after the host name).

This is the URL that was actually received by the web server before any rewriting took place, including the query string. Equivalent to the CGI variable REQUEST_URI.

uriWebwareRoot()

Return relative URL path of the Webware root location.

urlPath()

Get the URL path relative to the mount point, without query string.

This is actually the same as pathInfo().

For example, https://host/Webware/Context/Servlet?x=1 yields '/Context/Servlet'.

urlPathDir()

Same as urlPath, but only gives the directory.

For example, https://host/Webware/Context/Servlet?x=1 yields '/Context'.

value(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return the value with the given name.

Values are fields or cookies. Use this method when you're field/cookie agnostic.

writeExceptionReport(*handler*)

HTTPRequest.**htmlInfo**(*info*)

Return a single HTML string that represents the info structure.

Useful for inspecting objects via web browsers.

19.1.8 HTTPResponse

HTTP responses

class HTTPResponse.**HTTPResponse**(*transaction*, *strmOut*, *headers=None*)

Bases: `Response`

The base class for HTTP responses.

__init__(*transaction*, *strmOut*, *headers=None*)

Initialize the request.

addCookie(*cookie*)

Add a cookie that will be sent with this response.

cookie is a Cookie object instance. See the Cookie class docs.

assertNotCommitted()

Assert the the response is not already committed.

This raises a ConnectionError if the connection is already committed.

clearCookies()

Clear all the cookies.

clearHeaders()

Clear all the headers.

You might consider a setHeader('Content-Type', 'text/html') or something similar after this.

clearTransaction()

commit()

Commit response.

Write out all headers to the response stream, and tell the underlying response stream it can start sending data.

cookie(*name*)

Return the value of the specified cookie.

cookies()

Get all the cookies.

Returns a dictionary-style object of all Cookie objects that will be sent with this response.

delCookie(*name*, *path='/'*, *secure=False*)

Delete a cookie at the browser.

To do so, one has to create and send to the browser a cookie with parameters that will cause the browser to delete it.

delHeader(*name*)

Delete a specific header by name.

deliver()

Deliver response.

The final step in the processing cycle. Not used for much with responseStreams added.

displayError(*err*)

Display HTTPException errors, with status codes.

endTime()

flush(*autoFlush=True*)

Send all accumulated response data now.

Commits the response headers and tells the underlying stream to flush. if autoFlush is true, the responseStream will flush itself automatically from now on.

Caveat: Some web servers, especially IIS, will still buffer the output from your servlet until it terminates before transmitting the results to the browser. Also, server modules for Apache like mod_deflate or mod_gzip may do buffering of their own that will cause flush() to not result in data being sent immediately to the client. You can prevent this by setting a no-gzip note in the Apache configuration, e.g.

SetEnvIf Request_URI ^/Webware/MyServlet no-gzip=1

Even the browser may buffer its input before displaying it. For example, Netscape buffered text until it received an end-of-line or the beginning of a tag, and it didn't render tables until the end tag of the outermost table was seen. Some Firefox add-ons also buffer response data before it gets rendered. Some versions of MSIE will only start to display the page after they have received 256 bytes of output, so you may need to send extra whitespace before flushing to get MSIE to display the page.

`hasCookie`(*name*)

Return True if the specified cookie is present.

`hasHeader`(*name*)

`header`(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return the value of the specified header.

`headers`()

Return all the headers.

Returns a dictionary-style object of all header objects contained by this request.

`isCommitted`()

Check whether response is already committed.

Checks whether the response has already been partially or completely sent. If this method returns True, then no new headers/cookies can be added to the response anymore.

`mergeTextHeaders`(*headerStr*)

Merge text into our headers.

Given a string of headers (separated by newlines), merge them into our headers.

`protocol`()

Return the name and version of the protocol.

`rawResponse`()

Return the final contents of the response.

Don't invoke this method until after deliver().

Returns a dictionary representing the response containing only strings, numbers, lists, tuples, etc. with no backreferences. That means you don't need any special imports to examine the contents and you can marshal it. Currently there are two keys. 'headers' is list of tuples each of which contains two strings: the header and it's value. 'contents' is a string (that may be binary, for example, if an image were being returned).

`recordEndTime`()

Record the end time of the response.

Stores the current time as the end time of the response. This should be invoked at the end of deliver(). It may also be invoked by the application for those responses that never deliver due to an error.

`recordSession`()

Record session ID.

Invoked by commit() to record the session ID in the response (if a session exists). This implementation sets a cookie for that purpose. For people who don't like sweets, a future version could check a setting and instead of using cookies, could parse the HTML and update all the relevant URLs to include the session ID (which implies a big performance hit). Or we could require site developers to always pass their URLs through a function which adds the session ID (which implies pain). Personally, I'd rather just use cookies. You can experiment with different techniques by subclassing Session and overriding this method. Just make sure Application knows which "session" class to use.

It should be also considered to automatically add the server port to the cookie name in order to distinguish application instances running on different ports on the same server, or to use the port cookie-attribute introduced with RFC 2965 for that purpose.

`reset()`

Reset the response (such as headers, cookies and contents).

`sendError`(*code*, *msg=''*)

Set the status code to the specified code and message.

`sendRedirect`(*url*, *status=None*)

Redirect to another url.

This method sets the headers and content for the redirect, but does not change the cookies and other headers. Use clearCookies() or clearHeaders() as appropriate.

See https://www.ietf.org/rfc/rfc2616 (section 10.3.3) and https://www.ietf.org/rfc/rfc3875 (section 6.2.3).

`sendRedirectPermanent`(*url*)

Redirect permanently to another URL.

`sendRedirectSeeOther`(*url*)

Redirect to a URL that shall be retrieved with GET.

This method exists primarily to allow for the PRG pattern.

See https://en.wikipedia.org/wiki/Post/Redirect/Get

`sendRedirectTemporary`(*url*)

Redirect temporarily to another URL.

`setCookie`(*name*, *value*, *path='/'*, *expires='ONCLOSE'*, *secure=False*)

Set a cookie.

You can also set the path (which defaults to /).

You can also set when it expires. It can expire:

- 'NOW': this is the same as trying to delete it, but it doesn't really seem to work in IE
- 'ONCLOSE': the default behavior for cookies (expires when the browser closes)
- 'NEVER': some time in the far, far future.
- integer: a timestamp value
- tuple or struct_time: a tuple, as created by the time module
- datetime: a datetime.datetime object for the time (if without time zone, assumed to be *local*, not GMT time)
- timedelta: a duration counted from the present, e.g., datetime.timedelta(days=14) (2 weeks in the future)
- '+…': a time in the future, '…' should be something like 1w (1 week), 3h46m (3:45), etc. You can use y (year), b (month), w (week), d (day), h (hour), m (minute), s (second). This is done by the MiscUtils.DateInterval.

`setErrorHeaders`(*err*)

Set error headers for an HTTPException.

setHeader(*name*, *value*)

Set a specific header by name.

Parameters:
name: the header name value: the header value

setStatus(*code*, *msg=''*)

Set the status code of the response, such as 200, 'OK'.

size()

Return the size of the final contents of the response.

Don't invoke this method until after deliver().

streamOut()

write(*output=None*)

Write output to the response stream.

The output will be converted to a string, and then converted to bytes using the application output encoding, unless it is already bytes.

writeExceptionReport(*handler*)

writeHeaders()

Write headers to the response stream. Used internally.

19.1.9 HTTPServlet

HTTP servlets

class HTTPServlet.**HTTPServlet**

Bases: *Servlet*

A HTTP servlet.

HTTPServlet implements the respond() method to invoke methods such as respondToGet() and respondToPut() depending on the type of HTTP request. Example HTTP request methods are GET, POST, HEAD, etc. Subclasses implement HTTP method FOO in the Python method respondToFoo. Unsupported methods return a "501 Not Implemented" status.

Note that HTTPServlet inherits awake() and respond() methods from Servlet and that subclasses may make use of these.

See also: Servlet

__init__()

Subclasses must invoke super.

awake(*transaction*)

Send the awake message.

This message is sent to all objects that participate in the request-response cycle in a top-down fashion, prior to respond(). Subclasses must invoke super.

canBeReused()

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

`canBeThreaded()`

Return whether the servlet can be multithreaded.

This value should not change during the lifetime of the object. The default implementation returns False. Note: This is not currently used.

`close()`

`lastModified(`*_trans*`)`

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`log(`*message*`)`

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`name()`

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented(`*trans*`)`

`open()`

`respond(`*transaction*`)`

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, . . .).

`respondToHead(`*trans*`)`

Respond to a HEAD request.

A correct but inefficient implementation.

`runMethodForTransaction(`*transaction*, *method*, **args*, ***kw*`)`

`static runTransaction(`*transaction*`)`

`serverSidePath(`*path=None*`)`

Return the filesystem path of the page on the server.

`setFactory(`*factory*`)`

`sleep(`*transaction*`)`

Send the sleep message.

19.1.10 ImportManager

ImportManager

Manages imported modules and protects against concurrent imports.

Keeps lists of all imported Python modules and templates as well as other config files used by Webware for Python. Modules which are not directly imported can be monitored using hupper. This can be used to detect changes in source files, templates or config files in order to reload them automatically.

`class ImportManager.`**`ImportManager`**`(`**args*, ***kwargs*`)`

Bases: `object`

The import manager.

Keeps track of the Python modules and other system files that have been imported and are used by Webware.

`__init__``()`

Initialize import manager.

`delModules``(`*includePythonModules=False*, *excludePrefixes=None*`)`

Delete imported modules.

Deletes all the modules that have been imported unless they are part of Webware. This can be used to support auto reloading.

`fileList``(`*update=True*`)`

Return the list of tracked files.

`fileUpdated``(`*filename*, *update=True*, *getmtime=<function getmtime>*`)`

Check whether file has been updated.

`findSpec``(`*name*, *path*, *fullModuleName=None*`)`

Find the module spec for the given name at the given path.

`getReloader``()`

Get the current reloader if the application is monitored.

`moduleFromSpec``(`*spec*`)`

Load the module with the given module spec.

`notifyOfNewFiles``(`*hook*`)`

Register notification hook.

Called by someone else to register that they'd like to know when a new file is imported.

`recordFile``(`*filename*, *isfile=<function isfile>*`)`

Record a file.

`recordModule``(`*module*, *isfile=<function isfile>*`)`

Record a module.

`recordModules``(`*moduleNames=None*`)`

Record a list of modules (or all modules).

`updatedFile``(`*update=True*, *getmtime=<function getmtime>*`)`

Check whether one of the files has been updated.

`watchFile``(`*path*, *moduleName=None*, *getmtime=<function getmtime>*`)`

Add more files to watch without importing them.

19.1.11 JSONRPCServlet

JSON-RPC servlet base class

Written by Jean-Francois Pieronne

class JSONRPCServlet.**JSONRPCServlet**

Bases: *HTTPContent*

A superclass for Webware servlets using JSON-RPC techniques.

JSONRPCServlet can be used to make coding JSON-RPC applications easier.

Subclasses should override the method json_methods() which returns a list of method names. These method names refer to Webware Servlet methods that are able to be called by an JSON-RPC-enabled web page. This is very similar in functionality to Webware's actions.

Some basic security measures against JavaScript hijacking are taken by default which can be deactivated if you're not dealing with sensitive data. You can further increase security by adding shared secret mechanisms.

__init__()

Subclasses must invoke super.

actions()

The allowed actions.

Returns a list or a set of method names that are allowable actions from HTML forms. The default implementation returns []. See *_respond* for more about actions.

application()

The *Application* instance we're using.

awake(*transaction*)

Let servlet awake.

Makes instance variables from the transaction. This is where Page becomes unthreadsafe, as the page is tied to the transaction. This is also what allows us to implement functions like *write*, where you don't need to pass in the transaction or response.

callMethodOfServlet(*url*, *method*, **args*, ***kwargs*)

Call a method of another servlet.

See *Application.callMethodOfServlet* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

canBeReused()

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

canBeThreaded()

Declares whether servlet can be threaded.

Returns False because of the instance variables we set up in *awake*.

close()

`defaultAction()`

Default action.

The core method that gets called as a result of requests. Subclasses should override this.

`static endResponse()`

End response.

When this method is called during *awake* or *respond*, servlet processing will end immediately, and the accumulated response will be sent.

Note that *sleep* will still be called, providing a chance to clean up or free any resources.

`exposedMethods()`

Return a list or a set of all exposed RPC methods.

`forward(`*url*`)`

Forward request.

Forwards this request to another servlet. See *Application.forward* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`handleAction(`*action*`)`

Handle action.

Invoked by *_respond* when a legitimate action has been found in a form. Invokes *preAction*, the actual action method and *postAction*.

Subclasses rarely override this method.

`includeURL(`*url*`)`

Include output from other servlet.

Includes the response of another servlet in the current servlet's response. See *Application.includeURL* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`jsonCall()`

Execute method with arguments on the server side.

Returns JavaScript function to be executed by the client immediately.

`lastModified(`*_trans*`)`

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`log(`*message*`)`

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`methodNameForAction(`*name*`)`

Return method name for an action name.

Invoked by _respond() to determine the method name for a given action name which has been derived as the value of an `_action_` field. Since this is usually the label of an HTML submit button in a form, it is often needed to transform it in order to get a valid method name (for instance, blanks could be replaced by underscores and the like). This default implementation of the name transformation is the identity, it simply returns the name. Subclasses should override this method when action names don't match their method names; they could "mangle" the action names or look the method names up in a dictionary.

`name()`

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented(`*trans*`)`

`open()`

`outputEncoding()`

Get the default output encoding of the application.

`postAction(`*actionName*`)`

Things to do after action.

Invoked by self after invoking an action method. Subclasses may override to customize and may or may not invoke super as they see fit. The *actionName* is passed to this method, although it seems a generally bad idea to rely on this. However, it's still provided just in case you need that hook.

By default, this does nothing.

`preAction(`*actionName*`)`

Things to do before action.

Invoked by self prior to invoking an action method. The *actionName* is passed to this method, although it seems a generally bad idea to rely on this. However, it's still provided just in case you need that hook.

By default, this does nothing.

`request()`

The request (*HTTPRequest*) we're handling.

`respond(`*transaction*`)`

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, . . .).

`respondToGet(`*transaction*`)`

Respond to GET.

Invoked in response to a GET request method. All methods are passed to *_respond*.

`respondToHead(`*trans*`)`

Respond to a HEAD request.

A correct but inefficient implementation.

`respondToPost(`*transaction*`)`

Respond to POST.

Invoked in response to a POST request method. All methods are passed to *_respond*.

`response()`

The response (*HTTPResponse*) we're handling.

`runMethodForTransaction(`*transaction*, *method*, **args*, ***kw*`)`

`static runTransaction(`*transaction*`)` `sendRedirectAndEnd`(*url*, *status=None*)

Send redirect and end.

Sends a redirect back to the client and ends the response. This is a very popular pattern.

`sendRedirectPermanentAndEnd`(*url*)

Send permanent redirect and end.

`sendRedirectSeeOtherAndEnd`(*url*)

Send redirect to a URL to be retrieved with GET and end.

This is the proper method for the Post/Redirect/Get pattern.

`sendRedirectTemporaryAndEnd`(*url*)

Send temporary redirect and end.

`serverSidePath`(*path=None*)

Return the filesystem path of the page on the server.

`session`()

The session object.

This provides a state for the current user (associated with a browser instance, really). If no session exists, then a session will be created.

`sessionEncode`(*url=None*)

Utility function to access *Session.sessionEncode*.

Takes a url and adds the session ID as a parameter. This is for cases where you don't know if the client will accepts cookies.

`setFactory`(*factory*)

`sleep`(*transaction*)

Let servlet sleep again.

We unset some variables. Very boring.

`transaction`()

The *Transaction* we're currently handling.

static `urlDecode`(*s*)

Turn special % characters into actual characters.

This method does the same as the *urllib.unquote_plus()* function.

static `urlEncode`(*s*)

Quotes special characters using the % substitutions.

This method does the same as the *urllib.quote_plus()* function.

`write`(**args*)

Write to output.

Writes the arguments, which are turned to strings (with *str*) and concatenated before being written to the response. Unicode strings must be encoded before they can be written.

`writeError`(*msg*)

writeExceptionReport(*handler*)

Write extra information to the exception report.

The *handler* argument is the exception handler, and information is written there (using *writeTitle*, *write*, and *writeln*). This information is added to the exception report.

See *ExceptionHandler* for more information.

writeResult(*data*)

writeln(**args*)

Write to output with newline.

Writes the arguments (like *write*), adding a newline after. Unicode strings must be encoded before they can be written.

19.1.12 Page

The standard web page template.

class Page.**Page**

Bases: *HTTPContent*

The standard web page template.

Page is a type of HTTPContent that is more convenient for servlets which represent HTML pages generated in response to GET and POST requests. In fact, this is the most common type of Servlet.

Subclasses typically override `writeHeader`, `writeBody` and `writeFooter`.

They might also choose to override `writeHTML` entirely.

When developing a full-blown website, it's common to create a subclass of `Page` called `SitePage` which defines the common look and feel of the website and provides site-specific convenience methods. Then all other pages in your application then inherit from `SitePage`.

__init__()

Subclasses must invoke super.

actions()

The allowed actions.

Returns a list or a set of method names that are allowable actions from HTML forms. The default implementation returns []. See *_respond* for more about actions.

application()

The *Application* instance we're using.

awake(*transaction*)

Let servlet awake.

Makes instance variables from the transaction. This is where Page becomes unthreadsafe, as the page is tied to the transaction. This is also what allows us to implement functions like *write*, where you don't need to pass in the transaction or response.

callMethodOfServlet(*url*, *method*, **args*, ***kwargs*)

Call a method of another servlet.

See *Application.callMethodOfServlet* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`canBeReused()`

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

`canBeThreaded()`

Declares whether servlet can be threaded.

Returns False because of the instance variables we set up in *awake*.

`close()`

`defaultAction()`

The default action in a Page is to writeHTML().

`static endResponse()`

End response.

When this method is called during *awake* or *respond*, servlet processing will end immediately, and the accumulated response will be sent.

Note that *sleep* will still be called, providing a chance to clean up or free any resources.

`forward(`*url*`)`

Forward request.

Forwards this request to another servlet. See *Application.forward* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`handleAction(`*action*`)`

Handle action.

Invoked by *_respond* when a legitimate action has been found in a form. Invokes *preAction*, the actual action method and *postAction*.

Subclasses rarely override this method.

`htBodyArgs()`

The attributes for the <body> element.

Returns the arguments used for the HTML <body> tag. Invoked by writeBody().

With the prevalence of stylesheets (CSS), you can probably skip this particular HTML feature, but for historical reasons this sets the page to black text on white.

`htRootArgs()`

The attributes for the <html> element.

Returns the arguments used for the root HTML tag. Invoked by writeHTML() and preAction().

Authors are encouraged to specify a lang attribute, giving the document's language.

`htTitle()`

The page title as HTML.

Return self.title(). Subclasses sometimes override this to provide an HTML enhanced version of the title. This is the method that should be used when including the page title in the actual page contents.

static **htmlDecode**(*s*)

HTML decode special characters.

Alias for `WebUtils.Funcs.htmlDecode`. Decodes HTML entities.

static **htmlEncode**(*s*)

HTML encode special characters. Alias for `WebUtils.Funcs.htmlEncode`, quotes the special characters &, <, >, and "

includeURL(*url*)

Include output from other servlet.

Includes the response of another servlet in the current servlet's response. See *Application.includeURL* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

lastModified(*_trans*)

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

log(*message*)

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

methodNameForAction(*name*)

Return method name for an action name.

Invoked by _respond() to determine the method name for a given action name which has been derived as the value of an `_action_` field. Since this is usually the label of an HTML submit button in a form, it is often needed to transform it in order to get a valid method name (for instance, blanks could be replaced by underscores and the like). This default implementation of the name transformation is the identity, it simply returns the name. Subclasses should override this method when action names don't match their method names; they could "mangle" the action names or look the method names up in a dictionary.

name()

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

static **notImplemented**(*trans*)

open()

outputEncoding()

Get the default output encoding of the application.

postAction(*actionName*)

Things to do after actions.

Simply close the html tag (</html>).

preAction(*actionName*)

Things to do before actions.

For a page, we first writeDocType(), <html>, and then writeHead().

request()

The request (*HTTPRequest*) we're handling.

`respond`(*transaction*)

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, . . .).

`respondToGet`(*transaction*)

Respond to GET.

Invoked in response to a GET request method. All methods are passed to *_respond*.

`respondToHead`(*trans*)

Respond to a HEAD request.

A correct but inefficient implementation.

`respondToPost`(*transaction*)

Respond to POST.

Invoked in response to a POST request method. All methods are passed to *_respond*.

`response`()

The response (*HTTPResponse*) we're handling.

`runMethodForTransaction`(*transaction*, *method*, **args*, ***kw*)

`static runTransaction`(*transaction*)

`sendRedirectAndEnd`(*url*, *status=None*)

Send redirect and end.

Sends a redirect back to the client and ends the response. This is a very popular pattern.

`sendRedirectPermanentAndEnd`(*url*)

Send permanent redirect and end.

`sendRedirectSeeOtherAndEnd`(*url*)

Send redirect to a URL to be retrieved with GET and end.

This is the proper method for the Post/Redirect/Get pattern.

`sendRedirectTemporaryAndEnd`(*url*)

Send temporary redirect and end.

`serverSidePath`(*path=None*)

Return the filesystem path of the page on the server.

`session`()

The session object.

This provides a state for the current user (associated with a browser instance, really). If no session exists, then a session will be created.

`sessionEncode`(*url=None*)

Utility function to access *Session.sessionEncode*.

Takes a url and adds the session ID as a parameter. This is for cases where you don't know if the client will accepts cookies.

`setFactory`(*factory*)

`sleep`(*transaction*)

Let servlet sleep again.

We unset some variables. Very boring.

`title`()

The page title.

Subclasses often override this method to provide a custom title. This title should be absent of HTML tags. This implementation returns the name of the class, which is sometimes appropriate and at least informative.

`transaction`()

The *Transaction* we're currently handling.

static `urlDecode`(*s*)

Turn special % characters into actual characters.

This method does the same as the *urllib.unquote_plus()* function.

static `urlEncode`(*s*)

Quotes special characters using the % substitutions.

This method does the same as the *urllib.quote_plus()* function.

`write`(**args*)

Write to output.

Writes the arguments, which are turned to strings (with *str*) and concatenated before being written to the response. Unicode strings must be encoded before they can be written.

`writeBody`()

Write the <body> element of the page.

Writes the <body> portion of the page by writing the `<body>...</body>` (making use of `htBodyArgs`) and invoking `writeBodyParts` in between.

`writeBodyParts`()

Write the parts included in the <body> element.

Invokes `writeContent`. Subclasses should only override this method to provide additional page parts such as a header, sidebar and footer, that a subclass doesn't normally have to worry about writing.

For writing page-specific content, subclasses should override `writeContent` instead. This method is intended to be overridden by your SitePage.

See `SidebarPage` for an example override of this method.

Invoked by `writeBody`.

`writeContent`()

Write the unique, central content for the page.

Subclasses should override this method (not invoking super) to write their unique page content.

Invoked by `writeBodyParts`.

`writeDocType`()

Write the DOCTYPE tag.

Invoked by `writeHTML` to write the `<!DOCTYPE ...>` tag.

By default this gives the HTML 5 DOCTYPE.

Subclasses may override to specify something else.

`writeExceptionReport`(*handler*)

Write extra information to the exception report.

The *handler* argument is the exception handler, and information is written there (using *writeTitle*, *write*, and *writeln*). This information is added to the exception report.

See *ExceptionHandler* for more information.

`writeHTML`()

Write all the HTML for the page.

Subclasses may override this method (which is invoked by `_respond`) or more commonly its constituent methods, `writeDocType`, `writeHead` and `writeBody`.

You will want to override this method if:

- you want to format the entire HTML page yourself
- if you want to send an HTML page that has already been generated
- if you want to use a template that generates the entire page
- if you want to send non-HTML content; in this case, be sure to call self.response().setHeader('Content-Type', 'mime/type').

`writeHead`()

Write the <head> element of the page.

Writes the `<head>` portion of the page by writing the `<head>...</head>` tags and invoking `writeHeadParts` in between.

`writeHeadParts`()

Write the parts included in the <head> element.

Writes the parts inside the `<head>...</head>` tags. Invokes `writeTitle` and then `writeMetaData`, `writeStyleSheet` and `writeJavaScript`. Subclasses should override the `title` method and the three latter methods only.

`writeJavaScript`()

Write the JavaScript for the page.

This default implementation does nothing. Subclasses should override if necessary.

A typical implementation is:

```
self.writeln('<script src="ajax.js"></script>')
```

`writeMetaData`()

Write the meta data for the page.

This default implementation only specifies the output encoding. Subclasses should override if necessary.

`writeStyleSheet`()

Write the CSS for the page.

This default implementation does nothing. Subclasses should override if necessary.

A typical implementation is:

```
self.writeln('<link rel="stylesheet" href="StyleSheet.css">')
```

writeTitle()

Write the <title> element of the page.

Writes the `<title>` portion of the page. Uses `title`, which is where you should override.

writeln(**args*)

Write to output with newline.

Writes the arguments (like *write*), adding a newline after. Unicode strings must be encoded before they can be written.

19.1.13 PickleRPCServlet

Dict-RPC servlets.

class PickleRPCServlet.**PickleRPCServlet**

Bases: *RPCServlet*, *SafeUnpickler*

PickleRPCServlet is a base class for Dict-RPC servlets.

The “Pickle” refers to Python’s pickle module. This class is similar to XMLRPCServlet. By using Python pickles you get their convenience (assuming the client is Pythonic), but lose language independence. Some of us don’t mind that last one. ;-)

Conveniences over XML-RPC include the use of all of the following:

- Any pickle-able Python type (datetime for example)
- Python instances (aka objects)
- None
- Longs that are outside the 32-bit int boundaries
- Keyword arguments

Pickles should also be faster than XML, especially now that we support binary pickling and compression.

To make your own *PickleRPCServlet*, create a subclass and implement a method which is then named in *exposedMethods()*:

```
from PickleRPCServlet import PickleRPCServlet
class Math(PickleRPCServlet):
    def multiply(self, x, y):
        return x * y
    def exposedMethods(self):
        return ['multiply']
```

To make a PickleRPC call from another Python program, do this:

```
from MiscUtils.PickleRPC import Server
server = Server('http://localhost/Webware/Context/Math')
print(server.multiply(3, 4))     # 12
print(server.multiply('-', 10))  # ----------
```

If a request error is raised by the server, then *MiscUtils.PickleRPC.RequestError* is raised. If an unhandled exception is raised by the server, or the server response is malformed, then *MiscUtils.PickleRPC.ResponseError* (or one of its subclasses) is raised.

Tip: If you want callers of the RPC servlets to be able to introspect what methods are available, then include 'exposedMethods' in *exposedMethods()*.

If you wanted the actual response dictionary for some reason:

```
print(server._request('multiply', 3, 4))
# {'value': 12, 'timeReceived': ...}
```

In which case, an exception is not purposefully raised if the dictionary contains one. Instead, examine the dictionary.

For the dictionary formats and more information see the docs for *MiscUtils.PickleRPC*.

`__init__`()

Subclasses must invoke super.

`allowedGlobals`()

Allowed class names.

Must return a list of (moduleName, klassName) tuples for all classes that you want to allow to be unpickled.

Example:

```
return [('datetime', 'date')]
```

Allows datetime.date instances to be unpickled.

`awake`(*transaction*)

Begin transaction.

`call`(*methodName*, **args*, ***keywords*)

Call custom method.

Subclasses may override this class for custom handling of methods.

`canBeReused`()

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

`canBeThreaded`()

Return whether the servlet can be multithreaded.

This value should not change during the lifetime of the object. The default implementation returns False. Note: This is not currently used.

`close`()

`exposedMethods`()

Get exposed methods.

Subclasses should return a list of methods that will be exposed through XML-RPC.

`findGlobal`(*module*, *klass*)

Find class name.

`static handleException`(*transaction*)

Handle exception.

If ReportRPCExceptionsInWebware is set to True, then flush the response (because we don't want the standard HTML traceback to be appended to the response) and then handle the exception in the standard Webware way. This means logging it to the console, storing it in the error log, sending error email, etc. depending on the settings.

`lastModified`(*_trans*)

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`load`(*file*)

Unpickle a file.

`loads`(*s*)

Unpickle a string.

`log`(*message*)

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`name`()

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented`(*trans*)

`open`()

`respond`(*transaction*)

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, . . .).

`respondToHead`(*trans*)

Respond to a HEAD request.

A correct but inefficient implementation.

`respondToPost`(*trans*)

`resultForException`(*e*, *trans*)

Get text for exception.

Given an unhandled exception, returns the string that should be sent back in the RPC response as controlled by the RPCExceptionReturn setting.

`runMethodForTransaction`(*transaction*, *method*, **args*, ***kw*)

`static runTransaction`(*transaction*)

`static sendOK`(*contentType*, *contents*, *trans*, *contentEncoding=None*)

Send a 200 OK response with the given contents.

sendResponse(*trans*, *response*)

Timestamp the response dict and send it.

serverSidePath(*path=None*)

Return the filesystem path of the page on the server.

setFactory(*factory*)

sleep(*transaction*)

End transaction.

transaction()

Get the corresponding transaction.

Most uses of RPC will not need this.

static **useBinaryPickle**()

Determine whether binary pickling format shall be used.

When this returns True, the highest available binary pickling format will be used. Override this to return False to use the less-efficient text pickling format.

19.1.14 PlugIn

class PlugIn.**PlugIn**(*application*, *name*, *module*)

Bases: `object`

Template for Webware Plug-ins.

A plug-in is a software component that is loaded by Webware in order to provide additional Webware functionality without necessarily having to modify Webware's source. The most infamous plug-in is PSP (Python Server Pages) which ships with Webware.

Plug-ins often provide additional servlet factories, servlet subclasses, examples and documentation. Ultimately, it is the plug-in author's choice as to what to provide and in what manner.

Instances of this class represent plug-ins which are ultimately Python packages.

A plug-in must also be a Webware component which means that it will have a Properties.py file advertising its name, version, requirements, etc. You can ask a plug-in for its properties().

The plug-in/package must have an __init__.py which must contain the following function:

```
def installInWebware(application):
    ...
```

This function is invoked to take whatever actions are needed to plug the new component into Webware. See PSP for an example.

If you ask an Application for its plugIns(), you will get a list of instances of this class.

The path of the plug-in is added to sys.path, if it's not already there. This is convenient, but we may need a more sophisticated solution in the future to avoid name collisions between plug-ins.

Note that this class is hardly ever subclassed. The software in the plug-in package is what provides new functionality and there is currently no way to tell the Application to use custom subclasses of this class on a case-by-case basis (and so far there is currently no need).

Instructions for invoking:

```
# 'self' is typically Application. It gets passed to installInWebware()
p = PlugIn(self, 'Foo', '../Foo')
willNotLoadReason = plugIn.load()
if willNotLoadReason:
    print(f'Plug-in {path} cannot be loaded because:')
    print(willNotLoadReason)
    return None
p.install()
# Note that load() and install() could raise exceptions.
# You should expect this.
```

`__init__`(*application*, *name*, *module*)

Initializes the plug-in with basic information.

This lightweight constructor does not access the file system.

`directory`()

Return the directory in which the plug-in resides. Example: '..'

`examplePages`()

`examplePagesContext`()

`hasExamplePages`()

`install`()

Install plug-in by invoking its installInWebware() function.

`load`(*verbose=True*)

Loads the plug-in into memory, but does not yet install it.

Will return None on success, otherwise a message (string) that says why the plug-in could not be loaded.

`module`()

Return the Python module object of the plug-in.

`name`()

Return the name of the plug-in. Example: 'Foo'

`path`()

Return the full path of the plug-in. Example: '../Foo'

`properties`()

Return the properties.

This is a dictionary-like object, of the plug-in which comes from its Properties.py file. See MiscUtils.PropertiesObject.py.

`serverSidePath`(*path=None*)

`setUpExamplePages`()

Add a context for the examples.

exception `PlugIn.`**`PlugInError`**

Bases: `Exception`

Plug-in error.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

19.1.15 Properties

19.1.16 Request

An abstract request

class Request.**Request**

Bases: `object`

The abstract request class.

Request is a base class that offers the following:

- A time stamp (indicating when the request was made)
- An input stream
- Remote request information (address, name)
- Local host information (address, name, port)
- A security indicator

Request is an abstract class; developers typically use HTTPRequest.

__init__()

Initialize the request.

Subclasses are responsible for invoking super and initializing self._time.

clearTransaction()

input()

Return a file-style object that the contents can be read from.

isSecure()

Check whether this is a secure channel.

Returns true if request was made using a secure channel, such as HTTPS. This currently always returns false, since secure channels are not yet supported.

localAddress()

Get local address.

Returns a string containing the Internet Protocol (IP) address of the local host (e.g., the server) that received the request.

static localName()

Get local name.

Returns the fully qualified name of the local host (e.g., the server) that received the request.

localPort()

Get local port.

Returns the port of the local host (e.g., the server) that received the request.

remoteAddress()

Get the remote address.

Returns a string containing the Internet Protocol (IP) address of the client that sent the request.

remoteName()

Get the remote name.

Returns the fully qualified name of the client that sent the request, or the IP address of the client if the name cannot be determined.

responseClass()

Get the corresponding response class.

setTransaction(*trans*)

Set a transaction container.

time()

timeStamp()

Return time() as human readable string for logging and debugging.

transaction()

Get the transaction container.

writeExceptionReport(*handler*)

19.1.17 Response

An abstract response

class Response.**Response**(*trans*, *strmOut*)

Bases: `object`

The abstract response class.

Response is a base class that offers the following:

- A time stamp (indicating when the response was finished)
- An output stream

Response is an abstract class; developers typically use HTTPResponse.

__init__(*trans*, *strmOut*)

clearTransaction()

deliver()

endTime()

isCommitted()

recordEndTime()

Record the end time of the response.

Stores the current time as the end time of the response. This should be invoked at the end of deliver(). It may also be invoked by the application for those responses that never deliver due to an error.

reset()

streamOut()

write(*output*)

writeExceptionReport(*handler*)

19.1.18 RPCServlet

RPC servlets.

class RPCServlet.**RPCServlet**

Bases: *HTTPServlet*

RPCServlet is a base class for RPC servlets.

__init__()

Subclasses must invoke super.

awake(*transaction*)

Begin transaction.

call(*methodName*, **args*, ***keywords*)

Call custom method.

Subclasses may override this class for custom handling of methods.

canBeReused()

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

canBeThreaded()

Return whether the servlet can be multithreaded.

This value should not change during the lifetime of the object. The default implementation returns False. Note: This is not currently used.

close()

exposedMethods()

Get exposed methods.

Subclasses should return a list of methods that will be exposed through XML-RPC.

`static handleException`(*transaction*)

Handle exception.

If ReportRPCExceptionsInWebware is set to True, then flush the response (because we don't want the standard HTML traceback to be appended to the response) and then handle the exception in the standard Webware way. This means logging it to the console, storing it in the error log, sending error email, etc. depending on the settings.

`lastModified`(*_trans*)

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`log`(*message*)

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`name`()

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented`(*trans*)

`open`()

`respond`(*transaction*)

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, . . .).

`respondToHead`(*trans*)

Respond to a HEAD request.

A correct but inefficient implementation.

`resultForException`(*e*, *trans*)

Get text for exception.

Given an unhandled exception, returns the string that should be sent back in the RPC response as controlled by the RPCExceptionReturn setting.

`runMethodForTransaction`(*transaction*, *method*, **args*, ***kw*)

`static runTransaction`(*transaction*)

`static sendOK`(*contentType*, *contents*, *trans*, *contentEncoding=None*)

Send a 200 OK response with the given contents.

`serverSidePath`(*path=None*)

Return the filesystem path of the page on the server.

`setFactory`(*factory*)

`sleep`(*transaction*)

End transaction.

transaction()

Get the corresponding transaction.

Most uses of RPC will not need this.

19.1.19 Servlet

Abstract servlets

class Servlet.**Servlet**

Bases: `object`

A general servlet.

A servlet is a key portion of a server-based application that implements the semantics of a particular request by providing a response. This abstract class defines servlets at a very high level. Most often, developers will subclass HTTPServlet or even Page.

Servlets can be created once, then used and destroyed, or they may be reused several times over (it's up to the server). Therefore, servlet developers should take the proper actions in awake() and sleep() so that reuse can occur.

Objects that participate in a transaction include:

- Application
- Request
- Transaction
- Session
- Servlet
- Response

The awake(), respond() and sleep() methods form a message sandwich. Each is passed an instance of Transaction which gives further access to all the objects involved.

__init__()

Subclasses must invoke super.

awake(*transaction*)

Send the awake message.

This message is sent to all objects that participate in the request-response cycle in a top-down fashion, prior to respond(). Subclasses must invoke super.

canBeReused()

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

canBeThreaded()

Return whether the servlet can be multithreaded.

This value should not change during the lifetime of the object. The default implementation returns False. Note: This is not currently used.

close()

log(*message*)

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

name()

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

open()

respond(*transaction*)

Respond to a request.

runMethodForTransaction(*transaction*, *method*, **args*, ***kw*)

static **runTransaction**(*transaction*)

serverSidePath(*path=None*)

Return the filesystem path of the page on the server.

setFactory(*factory*)

sleep(*transaction*)

Send the sleep message.

19.1.20 ServletFactory

Servlet factory template.

class ServletFactory.**PythonServletFactory**(*application*)

Bases: *ServletFactory*

The factory for Python servlets.

This is the factory for ordinary Python servlets whose extensions are empty or .py. The servlets are unique per file since the file itself defines the servlet.

__init__(*application*)

Create servlet factory.

Stores a reference to the application in self._app, because subclasses may or may not need to talk back to the application to do their work.

extensions()

Return a list of extensions that match this handler.

Extensions should include the dot. An empty string indicates a file with no extension and is a valid value. The extension '.*' is a special case that is looked for a URL's extension doesn't match anything.

flushCache()

Flush the servlet cache and start fresh.

Servlets that are currently in the wild may find their way back into the cache (this may be a problem).

importAsPackage(*transaction*, *serverSidePathToImport*)

Import requested module.

Imports the module at the given path in the proper package/subpackage for the current request. For example, if the transaction has the URL http://localhost/Webware/MyContextDirectory/MySubdirectory/MyPage and path = 'some/random/path/MyModule.py' and the context is configured to have the name 'MyContext' then this function imports the module at that path as MyContext.MySubdirectory.MyModule . Note that the context name may differ from the name of the directory containing the context, even though they are usually the same by convention.

Note that the module imported may have a different name from the servlet name specified in the URL. This is used in PSP.

loadClass(*transaction*, *path*)

Load the appropriate class.

Given a transaction and a path, load the class for creating these servlets. Caching, pooling, and threadsafeness are all handled by servletForTransaction. This method is not expected to be threadsafe.

name()

Return the name of the factory.

This is a convenience for the class name.

returnServlet(*servlet*)

Return servlet to the pool.

Called by Servlet.close(), which returns the servlet to the servlet pool if necessary.

servletForTransaction(*transaction*)

Return a new servlet that will handle the transaction.

This method handles caching, and will call loadClass(trans, filepath) if no cache is found. Caching is generally controlled by servlets with the canBeReused() and canBeThreaded() methods.

uniqueness()

Return uniqueness type.

Returns a string to indicate the uniqueness of the ServletFactory's servlets. The Application needs to know if the servlets are unique per file, per extension or per application.

Return values are 'file', 'extension' and 'application'.

NOTE: Application so far only supports 'file' uniqueness.

class ServletFactory.**ServletFactory**(*application*)

Bases: `object`

Servlet factory template.

ServletFactory is an abstract class that defines the protocol for all servlet factories.

Servlet factories are used by the Application to create servlets for transactions.

A factory must inherit from this class and override uniqueness(), extensions() and either loadClass() or servletForTransaction(). Do not invoke the base class methods as they all raise AbstractErrors.

Each method is documented below.

`__init__`(*application*)

Create servlet factory.

Stores a reference to the application in self._app, because subclasses may or may not need to talk back to the application to do their work.

`extensions`()

Return a list of extensions that match this handler.

Extensions should include the dot. An empty string indicates a file with no extension and is a valid value. The extension '.*' is a special case that is looked for a URL's extension doesn't match anything.

`flushCache`()

Flush the servlet cache and start fresh.

Servlets that are currently in the wild may find their way back into the cache (this may be a problem).

`importAsPackage`(*transaction*, *serverSidePathToImport*)

Import requested module.

Imports the module at the given path in the proper package/subpackage for the current request. For example, if the transaction has the URL http://localhost/Webware/MyContextDirectory/MySubdirectory/MyPage and path = 'some/random/path/MyModule.py' and the context is configured to have the name 'MyContext' then this function imports the module at that path as MyContext.MySubdirectory.MyModule . Note that the context name may differ from the name of the directory containing the context, even though they are usually the same by convention.

Note that the module imported may have a different name from the servlet name specified in the URL. This is used in PSP.

`loadClass`(*transaction*, *path*)

Load the appropriate class.

Given a transaction and a path, load the class for creating these servlets. Caching, pooling, and threadsafeness are all handled by servletForTransaction. This method is not expected to be threadsafe.

`name`()

Return the name of the factory.

This is a convenience for the class name.

`returnServlet`(*servlet*)

Return servlet to the pool.

Called by Servlet.close(), which returns the servlet to the servlet pool if necessary.

`servletForTransaction`(*transaction*)

Return a new servlet that will handle the transaction.

This method handles caching, and will call loadClass(trans, filepath) if no cache is found. Caching is generally controlled by servlets with the canBeReused() and canBeThreaded() methods.

`uniqueness`()

Return uniqueness type.

Returns a string to indicate the uniqueness of the ServletFactory's servlets. The Application needs to know if the servlets are unique per file, per extension or per application.

Return values are 'file', 'extension' and 'application'.

NOTE: Application so far only supports 'file' uniqueness.

`ServletFactory.`**`iskeyword`**`()`

x.__contains__(y) <==> y in x.

19.1.21 Session

Implementation of client sessions.

`class` `Session.`**`Session`**`(`*trans*, *identifier=None*`)`

Bases: `object`

Implementation of client sessions.

All methods that deal with time stamps, such as creationTime(), treat time as the number of seconds since January 1, 1970.

Session identifiers are stored in cookies. Therefore, clients must have cookies enabled.

Note that the session id should be a string that is valid as part of a filename. This is currently true, and should be maintained if the session id generation technique is modified. Session ids can be used in filenames.

`__init__``(`*trans*, *identifier=None*`)`

`awake``(`*_trans*`)`

Let the session awake.

Invoked during the beginning of a transaction, giving a Session an opportunity to perform any required setup. The default implementation updates the 'lastAccessTime'.

`creationTime``()`

Return the time when this session was created.

`delValue``(`*name*`)`

`expiring``()`

Let the session expire.

Called when session is expired by the application. Subclasses should invoke super. Session store __delitem__()s should invoke if not isExpired().

`hasValue``(`*name*`)`

`identifier``()`

Return a string that uniquely identifies the session.

This method will create the identifier if needed.

`invalidate``()`

Invalidate the session.

It will be discarded the next time it is accessed.

`isDirty``()`

Check whether the session is dirty (has unsaved changes).

`isExpired``()`

Check whether the session has been previously expired.

See also: expiring()

`isNew()`

Check whether the session is new.

`lastAccessTime()`

Get last access time.

Returns the last time the user accessed the session through interaction. This attribute is updated in awake(), which is invoked at the beginning of a transaction.

`numTransactions()`

Get number of transactions.

Returns the number of transactions in which the session has been used.

`respond(`*trans*`)`

Let the session respond to a request.

The default implementation does nothing, but could do something in the future. Subclasses should invoke super.

`sessionEncode(`*url*`)`

Encode the session ID as a parameter to a url.

`setDirty(`*dirty=True*`)`

Set the dirty status of the session.

`setTimeout(`*timeout*`)`

Set the timeout on this session in seconds.

`setValue(`*name*, *value*`)`

`sleep(`*trans*`)`

Let the session sleep again.

Invoked during the ending of a transaction, giving a Session an opportunity to perform any required shutdown. The default implementation does nothing, but could do something in the future. Subclasses should invoke super.

`timeout()`

Return the timeout for this session in seconds.

`value(`*name*, *default=<class 'MiscUtils.NoDefault'>*`)`

`values()`

`writeExceptionReport(`*handler*`)`

exception `Session.SessionError`

Bases: `Exception`

Client session error

`__init__(`**args*, ***kwargs*`)`

`args`

`with_traceback()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

19.1.22 SessionDynamicStore

Session store using memory and files.

class SessionDynamicStore.**SessionDynamicStore**(*app*)

Bases: *SessionStore*

Stores the session in memory and in files.

To use this Session Store, set SessionStore in Application.config to 'Dynamic'. Other variables which can be set in Application.config are:

'MaxDynamicMemorySessions', which sets the maximum number of sessions that can be in memory at one time. Default is 10,000.

'DynamicSessionTimeout', which sets the default time for a session to stay in memory with no activity. Default is 15 minutes. When specifying this in Application.config, use minutes.

One-shot sessions (usually created by crawler bots) aren't moved to FileStore on periodical clean-up. They are still saved on SessionStore shutdown. This reduces the number of files in the Sessions directory.

__init__(*app*)

Create both a file and a memory store.

application()

Return the application owning the session store.

cleanStaleSessions(*task=None*)

Clean stale sessions.

Called by the Application to tell this store to clean out all sessions that have exceeded their lifetime. We want to have their native class functions handle it, though.

Ideally, intervalSweep would be run more often than the cleanStaleSessions functions for the actual stores. This may need to wait until we get the TaskKit in place, though.

The problem is the FileStore.cleanStaleSessions() method can take a while to run. So here, we only run the file sweep every fourth time.

clear()

Clear the session store in memory and remove all session files.

decoder()

Return the value deserializer for the store.

encoder()

Return the value serializer for the store.

get(*key*, *default=None*)

Return value if key available, else return the default.

has_key(*key*)

Check whether the session store has a given key.

intervalSweep()

The session sweeper interval function.

The interval function moves sessions from memory to file and can be run more often than the full cleanStaleSessions function.

items()
: Return a list with the (key, value) pairs for all sessions.

iteritems()
: Return an iterator over the (key, value) pairs for all sessions.

iterkeys()
: Return an iterator over the stored session keys.

itervalues()
: Return an iterator over the stored values of all sessions.

keys()
: Return a list with all keys of all the stored sessions.

memoryKeysInAccessTimeOrder()
: Fetch memory store's keys in ascending order of last access time.

moveToFile(*key*)
: Move the value for a session from memory to file.

moveToMemory(*key*)
: Move the value for a session from file to memory.

pop(*key*, *default=<class 'MiscUtils.NoDefault'>*)
: Return value if key available, else default (also remove key).

setEncoderDecoder(*encoder*, *decoder*)
: Set the serializer and deserializer for the store.

setdefault(*key*, *default=None*)
: Return value if key available, else default (also setting it).

storeAllSessions()
: Permanently save all sessions in the store.

storeSession(*session*)
: Save potentially changed session in the store.

values()
: Return a list with the values of all stored sessions.

19.1.23 SessionFileStore

Session store using files.

class SessionFileStore.**SessionFileStore**(*app*, *restoreFiles=None*)

Bases: `SessionStore`

A session file store.

Stores the sessions on disk in the Sessions/ directory, one file per session.

__init__(*app*, *restoreFiles=None*)
: Initialize the session file store.

  If restoreFiles is true, and sessions have been saved to file, the store will be initialized from these files.

`application()`
: Return the application owning the session store.

`cleanStaleSessions(`*_task=None*`)`
: Clean stale sessions.

  Called by the Application to tell this store to clean out all sessions that have exceeded their lifetime.

`clear()`
: Clear the session file store, removing all of the session files.

`decoder()`
: Return the value deserializer for the store.

`encoder()`
: Return the value serializer for the store.

`filenameForKey(`*key*`)`
: Return the name of the session file for the given key.

`get(`*key*, *default=None*`)`
: Return value if key available, else return the default.

`has_key(`*key*`)`
: Check whether the session store has a given key.

`items()`
: Return a list with the (key, value) pairs for all sessions.

`iteritems()`
: Return an iterator over the (key, value) pairs for all sessions.

`iterkeys()`
: Return an iterator over the stored session keys.

`itervalues()`
: Return an iterator over the stored values of all sessions.

`keys()`
: Return a list with the keys of all the stored sessions.

`pop(`*key*, *default=<class 'MiscUtils.NoDefault'>*`)`
: Return value if key available, else default (also remove key).

`removeKey(`*key*`)`
: Remove the session file for the given key.

`setEncoderDecoder(`*encoder*, *decoder*`)`
: Set the serializer and deserializer for the store.

`setdefault(`*key*, *default=None*`)`
: Return value if key available, else default (also setting it).

`storeAllSessions()`
: Permanently save all sessions in the store.

`storeSession(`*session*`)`
: Save session, writing it to the session file now.

`values()`
: Return a list with the values of all stored sessions.

19.1.24 SessionMemcachedStore

Session store using the Memcached memory object caching system.

class `SessionMemcachedStore.`**`SessionMemcachedStore`**`(`*app*`)`

Bases: *`SessionStore`*

A session store using Memcached.

Stores the sessions in a single Memcached store using ‘last write wins’ semantics. This increases fault tolerance and allows server clustering. In clustering configurations with concurrent writes for the same session(s) the last writer will always overwrite the session.

The keys are prefixed with a configurable namespace, allowing you to store other data in the same Memcached system.

Cleaning/timing out of sessions is performed by Memcached itself since no single application can know about the existence of all sessions or the last access for a given session. Besides it is built in Memcached functionality. Consequently, correct sizing of Memcached is necessary to hold all user’s session data.

Due to the way Memcached works, methods requiring access to the keys or for clearing the store do not work. You can configure whether you want to ignore such calls or raise an error in this case. By default, you will get a warning. It would be possible to emulate these functions by storing additional data in the memcache, such as a namespace counter or the number or even the full list of keys. However, if you are using more than one application instance, this would require fetching that data every time, since we cannot know whether another instance changed it. So we refrained from doing such sophisticated trickery and instead kept the implementation intentionally very simple and fast.

You need to install python-memcached to be able to use this module: https://www.tummy.com/software/python-memcached/ You also need a Memcached server: https://memcached.org/

Contributed by Steve Schwarz, March 2010. Small improvements by Christoph Zwerschke, April 2010.

`__init__``(`*app*`)`

Initialize the session store.

Subclasses must invoke super.

`application``()`

Return the application owning the session store.

`cleanStaleSessions``(`*_task=None*`)`

Clean stale sessions.

Memcached does this on its own, so we do nothing here.

`clear``()`

Clear the session store, removing all of its items.

Not supported by Memcached. We could emulate this by incrementing an additional namespace counter, but then we would need to fetch the current counter from the memcache before every access in order to keep different application instances in sync.

`decoder``()`

Return the value deserializer for the store.

`encoder``()`

Return the value serializer for the store.

`get``(`*key*, *default=None*`)`

Return value if key available, else return the default.

has_key(*key*)
Check whether the session store has a given key.

items()
Return a list with the (key, value) pairs for all sessions.

iteritems()
Return an iterator over the (key, value) pairs for all sessions.

iterkeys()
Return an iterator over the stored session keys.

itervalues()
Return an iterator over the stored values of all sessions.

keys()
Return a list with the keys of all the stored sessions.

Not supported by Memcached (see FAQ for explanation).

mcKey(*key*)
Create the real key with namespace to be used with Memcached.

pop(*key*, *default=<class 'MiscUtils.NoDefault'>*)
Return value if key available, else default (also remove key).

setEncoderDecoder(*encoder*, *decoder*)
Set the serializer and deserializer for the store.

setdefault(*key*, *default=None*)
Return value if key available, else default (also setting it).

storeAllSessions()
Permanently save all sessions in the store.

Should be used (only) when the application server is shut down. This closes the connection to the Memcached servers.

storeSession(*session*)
Save potentially changed session in the store.

values()
Return a list with the values of all stored sessions.

19.1.25 SessionMemoryStore

Session store in memory.

class SessionMemoryStore.**SessionMemoryStore**(*app*, *restoreFiles=None*)
Bases: `SessionStore`

Stores the session in memory as a dictionary.

This is fast and secure when you have one, persistent application instance.

__init__(*app*, *restoreFiles=None*)
Initialize the session memory store.

If restoreFiles is true, and sessions have been saved to file, the store will be initialized from these files.

`application()`
: Return the application owning the session store.

`cleanStaleSessions(`*_task=None*`)`
: Clean stale sessions.

  Called by the Application to tell this store to clean out all sessions that have exceeded their lifetime.

`clear()`
: Clear the session store, removing all of its items.

`decoder()`
: Return the value deserializer for the store.

`encoder()`
: Return the value serializer for the store.

`get(`*key*, *default=None*`)`
: Return value if key available, else return the default.

`has_key(`*key*`)`
: Check whether the session store has a given key.

`items()`
: Return a list with the (key, value) pairs for all sessions.

`iteritems()`
: Return an iterator over the (key, value) pairs for all sessions.

`iterkeys()`
: Return an iterator over the stored session keys.

`itervalues()`
: Return an iterator over the stored values of all sessions.

`keys()`
: Return a list with the keys of all the stored sessions.

`pop(`*key*, *default=<class 'MiscUtils.NoDefault'>*`)`
: Return value if key available, else default (also remove key).

`setEncoderDecoder(`*encoder*, *decoder*`)`
: Set the serializer and deserializer for the store.

`setdefault(`*key*, *default=None*`)`
: Return value if key available, else default (also setting it).

`storeAllSessions()`
: Permanently save all sessions in the store.

`storeSession(`*session*`)`
: Save already potentially changed session in the store.

`values()`
: Return a list with the values of all stored sessions.

19.1.26 SessionRedisStore

Session store using the Redis in-memory data store.

class `SessionRedisStore.`**`SessionRedisStore`**(*app*)

Bases: *SessionStore*

A session store using Redis.

Stores the sessions in a single Redis store using 'last write wins' semantics. This increases fault tolerance and allows server clustering. In clustering configurations with concurrent writes for the same session(s) the last writer will always overwrite the session.

The keys are prefixed with a configurable namespace, allowing you to store other data in the same Redis system.

Cleaning/timing out of sessions is performed by Redis itself since no single application can know about the existence of all sessions or the last access for a given session. Besides it is built in Redis functionality. Consequently, correct sizing of Redis is necessary to hold all user's session data.

You need to install the redis client to be able to use this module: https://pypi.python.org/pypi/redis You also need a Redis server: https://redis.io/

Contributed by Christoph Zwerschke, August 2016.

`__init__`(*app*)

Initialize the session store.

Subclasses must invoke super.

`application`()

Return the application owning the session store.

`cleanStaleSessions`(*_task=None*)

Clean stale sessions.

Redis does this on its own, so we do nothing here.

`clear`()

Clear the session store, removing all of its items.

`decoder`()

Return the value deserializer for the store.

`encoder`()

Return the value serializer for the store.

`get`(*key*, *default=None*)

Return value if key available, else return the default.

`has_key`(*key*)

Check whether the session store has a given key.

`items`()

Return a list with the (key, value) pairs for all sessions.

`iteritems`()

Return an iterator over the (key, value) pairs for all sessions.

`iterkeys`()

Return an iterator over the stored session keys.

itervalues()

Return an iterator over the stored values of all sessions.

keys()

Return a list with the keys of all the stored sessions.

pop(*key*, *default=<class 'MiscUtils.NoDefault'>*)

Return value if key available, else default (also remove key).

redisKey(*key*)

Create the real key with namespace to be used with Redis.

setEncoderDecoder(*encoder*, *decoder*)

Set the serializer and deserializer for the store.

setdefault(*key*, *default=None*)

Return value if key available, else default (also setting it).

storeAllSessions()

Permanently save all sessions in the store.

Should be used (only) when the application server is shut down. This closes the connections to the Redis server.

storeSession(*session*)

Save potentially changed session in the store.

values()

Return a list with the values of all stored sessions.

19.1.27 SessionShelveStore

Session store using the shelve module.

class SessionShelveStore.**SessionShelveStore**(*app*, *restoreFiles=None*, *filename=None*)

Bases: `SessionStore`

A session store implemented with a shelve object.

To use this store, set SessionStore in Application.config to 'Shelve'.

__init__(*app*, *restoreFiles=None*, *filename=None*)

Initialize the session shelf.

If restoreFiles is true, existing shelve file(s) will be reused.

application()

Return the application owning the session store.

cleanStaleSessions(*task=None*)

Clean stale sessions.

clear()

Clear the session store, removing all of its items.

decoder()

Return the value deserializer for the store.

encoder()

Return the value serializer for the store.

get(*key*, *default=None*)

Return value if key available, else return the default.

has_key(*key*)

Check whether the session store has a given key.

intervalSweep()

The session sweeper interval function.

items()

Return a list with the (key, value) pairs for all sessions.

iteritems()

Return an iterator over the (key, value) pairs for all sessions.

iterkeys()

Return an iterator over the stored session keys.

itervalues()

Return an iterator over the stored values of all sessions.

keys()

Return a list with the keys of all the stored sessions.

pop(*key*, *default=<class 'MiscUtils.NoDefault'>*)

Return value if key available, else default (also remove key).

setEncoderDecoder(*encoder*, *decoder*)

Set the serializer and deserializer for the store.

setdefault(*key*, *default=None*)

Return value if key available, else default (also setting it).

storeAllSessions()

Permanently save all sessions in the store.

Should be used (only) when the application server is shut down.

storeSession(*session*)

Save potentially changed session in the store.

values()

Return a list with the values of all stored sessions.

19.1.28 SessionStore

A general session store.

class SessionStore.**SessionStore**(*app*)

Bases: `object`

A general session store.

SessionStores are dictionary-like objects used by Application to store session state. This class is abstract and it's up to the concrete subclass to implement several key methods that determine how sessions are stored (such as in memory, on disk or in a database). We assume that session keys are always strings.

Subclasses often encode sessions for storage somewhere. In light of that, this class also defines methods encoder(), decoder() and setEncoderDecoder(). The encoder and decoder default to the load() and dump() functions of the pickle module. However, using the setEncoderDecoder() method, you can use the functions from marshal (if appropriate) or your own encoding scheme. Subclasses should use encoder() and decoder() (and not pickle.load() and pickle.dump()).

Subclasses may rely on the attribute self._app to point to the application.

Subclasses should be named SessionFooStore since Application expects "Foo" to appear for the "SessionStore" setting and automatically prepends Session and appends Store. Currently, you will also need to add another import statement in Application.py. Search for SessionStore and you'll find the place.

TO DO

- Should there be a check-in/check-out strategy for sessions to prevent concurrent requests on the same session? If so, that can probably be done at this level (as opposed to pushing the burden on various subclasses).

`__init__`(*app*)

Initialize the session store.

Subclasses must invoke super.

`application`()

Return the application owning the session store.

`cleanStaleSessions`(*_task=None*)

Clean stale sessions.

Called by the Application to tell this store to clean out all sessions that have exceeded their lifetime.

`clear`()

Clear the session store, removing all of its items.

Subclasses must implement this method.

`decoder`()

Return the value deserializer for the store.

`encoder`()

Return the value serializer for the store.

`get`(*key*, *default=None*)

Return value if key available, else return the default.

`has_key`(*key*)

Check whether the session store has a given key.

`items`()

Return a list with the (key, value) pairs for all sessions.

`iteritems`()

Return an iterator over the (key, value) pairs for all sessions.

`iterkeys`()

Return an iterator over the stored session keys.

`itervalues`()

Return an iterator over the stored values of all sessions.

keys()

Return a list with the keys of all the stored sessions.

Subclasses must implement this method.

pop(*key*, *default=None*)

Return value if key available, else default (also remove key).

Subclasses must implement this method.

setEncoderDecoder(*encoder*, *decoder*)

Set the serializer and deserializer for the store.

setdefault(*key*, *default=None*)

Return value if key available, else default (also setting it).

Subclasses must implement this method.

storeAllSessions()

Permanently save all sessions in the store.

Used when the application server is shut down.

Subclasses must implement this method.

storeSession(*session*)

Save potentially changed session in the store.

Used at the end of transactions.

Subclasses must implement this method.

values()

Return a list with the values of all stored sessions.

SessionStore.**dumpWithHighestProtocol**(*obj*, *f*)

Same as pickle.dump, but by default with the highest protocol.

19.1.29 SidebarPage

Webware page template class for pages with a sidebar.

class SidebarPage.**SidebarPage**

Bases: *Page*

Webware page template class for pages with a sidebar.

SidebarPage is an abstract superclass for pages that have a sidebar (as well as a header and "content well"). Sidebars are normally used for navigation (e.g., a menu or list of links), showing small bits of info and occasionally a simple form (such as login or search).

Subclasses should override cornerTitle(), writeSidebar() and writeContent() (and title() if necessary; see Page).

The utility methods menuHeading() and menuItem() can be used by subclasses, typically in their implementation of writeSidebar().

Webware itself uses this class: Examples/ExamplePage and Admin/AdminPage both inherit from it.

__init__()

Subclasses must invoke super.

`actions()`

The allowed actions.

Returns a list or a set of method names that are allowable actions from HTML forms. The default implementation returns []. See *_respond* for more about actions.

`application()`

The *Application* instance we're using.

`awake(`*transaction*`)`

Let servlet awake.

Makes instance variables from the transaction. This is where Page becomes unthreadsafe, as the page is tied to the transaction. This is also what allows us to implement functions like *write*, where you don't need to pass in the transaction or response.

`callMethodOfServlet(`*url*, *method*, **args*, ***kwargs*`)`

Call a method of another servlet.

See *Application.callMethodOfServlet* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`canBeReused()`

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

`canBeThreaded()`

Declares whether servlet can be threaded.

Returns False because of the instance variables we set up in *awake*.

`close()`

`cornerTitle()`

`defaultAction()`

The default action in a Page is to writeHTML().

`static endResponse()`

End response.

When this method is called during *awake* or *respond*, servlet processing will end immediately, and the accumulated response will be sent.

Note that *sleep* will still be called, providing a chance to clean up or free any resources.

`forward(`*url*`)`

Forward request.

Forwards this request to another servlet. See *Application.forward* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`handleAction(`*action*`)`

Handle action.

Invoked by *_respond* when a legitimate action has been found in a form. Invokes *preAction*, the actual action method and *postAction*.

Subclasses rarely override this method.

`htBodyArgs()`

The attributes for the <body> element.

Returns the arguments used for the HTML <`body`> tag. Invoked by writeBody().

With the prevalence of stylesheets (CSS), you can probably skip this particular HTML feature, but for historical reasons this sets the page to black text on white.

`htRootArgs()`

The attributes for the <html> element.

Returns the arguments used for the root HTML tag. Invoked by writeHTML() and preAction().

Authors are encouraged to specify a lang attribute, giving the document's language.

`htTitle()`

The page title as HTML.

Return self.title(). Subclasses sometimes override this to provide an HTML enhanced version of the title. This is the method that should be used when including the page title in the actual page contents.

`static htmlDecode(`*s*`)`

HTML decode special characters.

Alias for `WebUtils.Funcs.htmlDecode`. Decodes HTML entities.

`static htmlEncode(`*s*`)`

HTML encode special characters. Alias for `WebUtils.Funcs.htmlEncode`, quotes the special characters &, <, >, and "

`includeURL(`*url*`)`

Include output from other servlet.

Includes the response of another servlet in the current servlet's response. See *Application.includeURL* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`lastModified(`*_trans*`)`

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`log(`*message*`)`

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`menuHeading(`*title*`)`

`menuItem(`*title*, *url=None*, *suffix=None*, *indentLevel=1*`)`

`methodNameForAction(`*name*`)`

Return method name for an action name.

Invoked by _respond() to determine the method name for a given action name which has been derived as the value of an `_action_` field. Since this is usually the label of an HTML submit button in a form, it is often needed to transform it in order to get a valid method name (for instance, blanks could be replaced by underscores and the like). This default implementation of the name transformation is the identity, it simply returns the name. Subclasses should override this method when action names don't match their method names; they could "mangle" the action names or look the method names up in a dictionary.

`name()`

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented(`*trans*`)`

`open()`

`outputEncoding()`

Get the default output encoding of the application.

`postAction(`*actionName*`)`

Things to do after actions.

Simply close the html tag (</html>).

`preAction(`*actionName*`)`

Things to do before actions.

For a page, we first writeDocType(), <html>, and then writeHead().

`request()`

The request (*HTTPRequest*) we're handling.

`respond(`*transaction*`)`

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, …).

`respondToGet(`*transaction*`)`

Respond to GET.

Invoked in response to a GET request method. All methods are passed to *_respond*.

`respondToHead(`*trans*`)`

Respond to a HEAD request.

A correct but inefficient implementation.

`respondToPost(`*transaction*`)`

Respond to POST.

Invoked in response to a POST request method. All methods are passed to *_respond*.

`response()`

The response (*HTTPResponse*) we're handling.

`runMethodForTransaction(`*transaction*, *method*, **args*, ***kw*`)`

`static runTransaction(`*transaction*`)`

`sendRedirectAndEnd(`*url*, *status=None*`)`

Send redirect and end.

Sends a redirect back to the client and ends the response. This is a very popular pattern.

`sendRedirectPermanentAndEnd(`*url*`)`

Send permanent redirect and end.

`sendRedirectSeeOtherAndEnd`(*url*)

Send redirect to a URL to be retrieved with GET and end.

This is the proper method for the Post/Redirect/Get pattern.

`sendRedirectTemporaryAndEnd`(*url*)

Send temporary redirect and end.

`serverSidePath`(*path=None*)

Return the filesystem path of the page on the server.

`session`()

The session object.

This provides a state for the current user (associated with a browser instance, really). If no session exists, then a session will be created.

`sessionEncode`(*url=None*)

Utility function to access *Session.sessionEncode*.

Takes a url and adds the session ID as a parameter. This is for cases where you don't know if the client will accepts cookies.

`setFactory`(*factory*)

`sleep`(*transaction*)

Let servlet sleep again.

We unset some variables. Very boring.

`title`()

The page title.

Subclasses often override this method to provide a custom title. This title should be absent of HTML tags. This implementation returns the name of the class, which is sometimes appropriate and at least informative.

`transaction`()

The *Transaction* we're currently handling.

static `urlDecode`(*s*)

Turn special % characters into actual characters.

This method does the same as the *urllib.unquote_plus()* function.

static `urlEncode`(*s*)

Quotes special characters using the % substitutions.

This method does the same as the *urllib.quote_plus()* function.

`write`(**args*)

Write to output.

Writes the arguments, which are turned to strings (with *str*) and concatenated before being written to the response. Unicode strings must be encoded before they can be written.

`writeBanner`()

`writeBody`()

Write the <body> element of the page.

Writes the <body> portion of the page by writing the `<body>...</body>` (making use of `htBodyArgs`) and invoking `writeBodyParts` in between.

`writeBodyParts()`

Write the parts included in the <body> element.

Invokes `writeContent`. Subclasses should only override this method to provide additional page parts such as a header, sidebar and footer, that a subclass doesn't normally have to worry about writing.

For writing page-specific content, subclasses should override `writeContent` instead. This method is intended to be overridden by your SitePage.

See `SidebarPage` for an example override of this method.

Invoked by `writeBody`.

`writeContent()`

Write the unique, central content for the page.

Subclasses should override this method (not invoking super) to write their unique page content.

Invoked by `writeBodyParts`.

`writeContextsMenu()`

`writeDocType()`

Write the DOCTYPE tag.

Invoked by `writeHTML` to write the `<!DOCTYPE ...>` tag.

By default this gives the HTML 5 DOCTYPE.

Subclasses may override to specify something else.

`writeExceptionReport(`*handler*`)`

Write extra information to the exception report.

The *handler* argument is the exception handler, and information is written there (using *writeTitle*, *write*, and *writeln*). This information is added to the exception report.

See *ExceptionHandler* for more information.

`writeHTML()`

Write all the HTML for the page.

Subclasses may override this method (which is invoked by `_respond`) or more commonly its constituent methods, `writeDocType`, `writeHead` and `writeBody`.

You will want to override this method if:

- you want to format the entire HTML page yourself
- if you want to send an HTML page that has already been generated
- if you want to use a template that generates the entire page
- if you want to send non-HTML content; in this case, be sure to call self.response().setHeader('Content-Type', 'mime/type').

`writeHead()`

Write the <head> element of the page.

Writes the `<head>` portion of the page by writing the `<head>...</head>` tags and invoking `writeHeadParts` in between.

writeHeadParts()

Write the parts included in the <head> element.

Writes the parts inside the `<head>...</head>` tags. Invokes `writeTitle` and then `writeMetaData`, `writeStyleSheet` and `writeJavaScript`. Subclasses should override the `title` method and the three latter methods only.

writeJavaScript()

Write the JavaScript for the page.

This default implementation does nothing. Subclasses should override if necessary.

A typical implementation is:

```
self.writeln('<script src="ajax.js"></script>')
```

writeMetaData()

Write the meta data for the page.

This default implementation only specifies the output encoding. Subclasses should override if necessary.

writeSidebar()

writeStyleSheet()

We're using a simple internal style sheet.

This way we avoid having to care about where an external style sheet should be located when this class is used in another context.

writeTitle()

Write the <title> element of the page.

Writes the `<title>` portion of the page. Uses `title`, which is where you should override.

writeVersions()

writeWebwareDocsMenu()

writeWebwareExitsMenu()

writeWebwareSidebarSections()

Write sidebar sections.

This method (and consequently the methods it invokes) are provided for Webware's example and admin pages. It writes sections such as contexts, e-mails, exits and versions.

writeln(**args*)

Write to output with newline.

Writes the arguments (like *write*), adding a newline after. Unicode strings must be encoded before they can be written.

19.1.30 Transaction

The Transaction container.

class `Transaction.`**`Transaction`**(*application*, *request=None*)

Bases: `object`

The Transaction container.

A transaction serves as:

- A container for all objects involved in the transaction. The objects include application, request, response, session and servlet.
- A message dissemination point. The messages include awake(), respond() and sleep().

When first created, a transaction has no session. However, it will create or retrieve one upon being asked for session().

The life cycle of a transaction begins and ends with Application's dispatchRequest().

`__init__`(*application*, *request=None*)

`application`()

Get the corresponding application.

`awake`()

Send awake() to the session (if there is one) and the servlet.

Currently, the request and response do not partake in the awake()-respond()-sleep() cycle. This could definitely be added in the future if any use was demonstrated for it.

`die`()

End transaction.

This method should be invoked when the entire transaction is finished with. Currently, this is invoked by the Application. This method removes references to the different objects in the transaction, breaking cyclic reference chains and speeding up garbage collection.

`dump`(*file=None*)

Dump debugging info to stdout.

`duration`()

Return the duration, in seconds, of the transaction.

This is basically the response end time minus the request start time.

`error`()

Return Exception instance if there was any.

`errorOccurred`()

Check whether a server error occurred.

`hasSession`()

Return true if the transaction has a session.

`request`()

Get the corresponding request.

`respond`()

Respond to the request.

response()

Get the corresponding response.

servlet()

Return the current servlet that is processing.

Remember that servlets can be nested.

session()

Return the session for the transaction.

A new transaction is created if necessary. Therefore, this method never returns None. Use hasSession() if you want to find out if a session already exists.

setError(*err*)

Set Exception instance.

Invoked by the application if an Exception is raised to the application level.

setResponse(*response*)

Set the corresponding response.

setServlet(*servlet*)

Set the servlet for processing the transaction.

setSession(*session*)

Set the session for the transaction.

sleep()

Send sleep() to the session and the servlet.

Note that sleep() is sent in reverse order as awake() (which is typical for shutdown/cleanup methods).

writeExceptionReport(*handler*)

Write extra information to the exception report.

19.1.31 UnknownFileTypeServlet

Servlet factory for unknown file types.

class UnknownFileTypeServlet.**UnknownFileTypeServlet**(*application*)

Bases: *HTTPServlet*, *Configurable*

Servlet for unknown file types.

Normally this class is just a "private" utility class for Webware's purposes. However, you may find it useful to subclass on occasion, such as when the server side file path is determined by something other than a direct correlation to the URL. Here is such an example:

from UnknownFileTypeServlet import UnknownFileTypeServlet import os

class Image(UnknownFileTypeServlet):

imageDir = '/var/images'

def filename(self, trans):
filename = trans.request().field('i') filename = os.path.join(self.imageDir, filename) return filename

`__init__`(*application*)

Subclasses must invoke super.

`awake`(*transaction*)

Send the awake message.

This message is sent to all objects that participate in the request-response cycle in a top-down fashion, prior to respond(). Subclasses must invoke super.

`canBeReused`()

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

`canBeThreaded`()

Return whether the servlet can be multithreaded.

This value should not change during the lifetime of the object. The default implementation returns False. Note: This is not currently used.

`close`()

`commandLineConfig`()

Return the settings that came from the command-line.

These settings come via addCommandLineSetting().

`config`()

Return the configuration of the object as a dictionary.

This is a combination of defaultConfig() and userConfig(). This method caches the config.

`configFilename`()

Return the full name of the user config file.

Users can override the configuration by this config file. Subclasses must override to specify a name. Returning None is valid, in which case no user config file will be loaded.

`configName`()

Return the name of the configuration file without the extension.

This is the portion of the config file name before the '.config'. This is used on the command-line.

`configReplacementValues`()

Return a dictionary for substitutions in the config file.

This must be a dictionary suitable for use with "string % dict" that should be used on the text in the config file. If an empty dictionary (or None) is returned, then no substitution will be attempted.

`defaultConfig`()

Get the default config.

Taken from Application's 'UnknownFileTypes' default setting.

`filename`(*trans*)

Return the filename to be served.

A subclass could override this in order to serve files from other disk locations based on some logic.

`hasSetting`(*name*)

Check whether a configuration setting has been changed.

`lastModified`(*trans*)

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`log`(*message*)

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`name`()

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented`(*trans*)

`open`()

`printConfig`(*dest=None*)

Print the configuration to the given destination.

The default destination is stdout. A fixed with font is assumed for aligning the values to start at the same column.

`static readConfig`(*filename*)

Read the configuration from the file with the given name.

Raises an UIError if the configuration cannot be read.

This implementation assumes the file is stored in utf-8 encoding with possible BOM at the start, but also tries to read as latin-1 if it cannot be decoded as utf-8. Subclasses can override this behavior.

`static redirectSansScript`(*trans*)

Redirect to web server.

Sends a redirect to a URL that doesn't contain the script name. Under the right configuration, this will cause the web server to then be responsible for the URL rather than the WSGI server. Keep in mind that links off the target page will *not* include the script name in the URL.

`respond`(*transaction*)

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, …).

`respondToGet`(*trans*)

Respond to GET request.

Responds to the transaction by invoking self.foo() for foo is specified by the 'Technique' setting.

`respondToHead`(*trans*)

Respond to GET request.

Responds to the transaction by invoking self.foo() for foo is specified by the 'Technique' setting.

`respondToPost`(*trans*)

Respond to POST request.

Invoke self.respondToGet().

Since posts are usually accompanied by data, this might not be the best policy. However, a POST would most likely be for a CGI, which currently no one is mixing in with their Webware-based web sites.

`runMethodForTransaction`(*transaction*, *method*, **args*, ***kw*)

`static runTransaction`(*transaction*)

`serveContent`(*trans*)

`serverSidePath`(*path=None*)

Return the filesystem path of the page on the server.

`setFactory`(*factory*)

`setSetting`(*name*, *value*)

Set a particular configuration setting.

`setting`(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return the value of a particular setting in the configuration.

`shouldCacheContent`()

Return whether the content should be cached or not.

Returns a boolean that controls whether or not the content served through this servlet is cached. The default behavior is to return the CacheContent setting. Subclasses may override to always True or False, or incorporate some other logic.

`sleep`(*transaction*)

Send the sleep message.

`userConfig`()

Get the user config.

Taken from Application's 'UnknownFileTypes' user setting.

`static validTechniques`()

`class UnknownFileTypeServlet.UnknownFileTypeServletFactory`(*application*)

Bases: `ServletFactory`

The servlet factory for unknown file types.

I.e. all files other than .py, .psp and the other types we support.

`__init__`(*application*)

Create servlet factory.

Stores a reference to the application in self._app, because subclasses may or may not need to talk back to the application to do their work.

`extensions`()

Return a list of extensions that match this handler.

Extensions should include the dot. An empty string indicates a file with no extension and is a valid value. The extension '.*' is a special case that is looked for a URL's extension doesn't match anything.

`flushCache()`

Flush the servlet cache and start fresh.

Servlets that are currently in the wild may find their way back into the cache (this may be a problem).

`importAsPackage(`*transaction*, *serverSidePathToImport*`)`

Import requested module.

Imports the module at the given path in the proper package/subpackage for the current request. For example, if the transaction has the URL http://localhost/Webware/MyContextDirectory/MySubdirectory/MyPage and path = 'some/random/path/MyModule.py' and the context is configured to have the name 'MyContext' then this function imports the module at that path as MyContext.MySubdirectory.MyModule . Note that the context name may differ from the name of the directory containing the context, even though they are usually the same by convention.

Note that the module imported may have a different name from the servlet name specified in the URL. This is used in PSP.

`loadClass(`*transaction*, *path*`)`

Load the appropriate class.

Given a transaction and a path, load the class for creating these servlets. Caching, pooling, and threadsafeness are all handled by servletForTransaction. This method is not expected to be threadsafe.

`name()`

Return the name of the factory.

This is a convenience for the class name.

`returnServlet(`*servlet*`)`

Return servlet to the pool.

Called by Servlet.close(), which returns the servlet to the servlet pool if necessary.

`servletForTransaction(`*transaction*`)`

Return a new servlet that will handle the transaction.

This method handles caching, and will call loadClass(trans, filepath) if no cache is found. Caching is generally controlled by servlets with the canBeReused() and canBeThreaded() methods.

`uniqueness()`

Return uniqueness type.

Returns a string to indicate the uniqueness of the ServletFactory's servlets. The Application needs to know if the servlets are unique per file, per extension or per application.

Return values are 'file', 'extension' and 'application'.

NOTE: Application so far only supports 'file' uniqueness.

19.1.32 URLParser

URLParser

URL parsing is done through objects which are subclasses of the *URLParser* class. *Application* delegates most of the URL parsing to these objects.

Application has a single "root" URL parser, which is used to parse all URLs. This parser then can pass the request on to other parsers, usually taking off parts of the URL with each step.

This root parser is generally *ContextParser*, which is instantiated and set up by *Application* (accessible through *Application.rootURLParser*).

class `URLParser.`**`ContextParser`**(*app*)

Bases: *`URLParser`*

Find the context of a request.

ContextParser uses the `Application.config` context settings to find the context of the request. It then passes the request to a FileParser rooted in the context path.

The context is the first element of the URL, or if no context matches that then it is the `default` context (and the entire URL is passed to the default context's FileParser).

There is generally only one ContextParser, which can be found as `application.rootURLParser()`.

`__init__`(*app*)

Create ContextParser.

ContextParser is usually created by Application, which passes all requests to it.

In __init__ we take the `Contexts` setting from Application.config and parse it slightly.

`absContextPath`(*path*)

Get absolute context path.

Resolves relative paths, which are assumed to be relative to the Application's serverSidePath (the working directory).

`addContext`(*name*, *path*)

Add a context to the system.

The context will be imported as a package, going by *name*, from the given directory path. The directory doesn't have to match the context name.

`findServletForTransaction`(*trans*)

Returns a servlet for the transaction.

This is the top-level entry point, below it *parse* is used.

`parse`(*trans*, *requestPath*)

Parse request.

Get the context name, and dispatch to a FileParser rooted in the context's path.

The context name and file path are stored in the request (accessible through *Request.serverSidePath* and *Request.contextName*).

`resolveDefaultContext`(*dest*)

Find default context.

Figure out if the default context refers to an existing context, the same directory as an existing context, or a unique directory.

Returns the name of the context that the default context refers to, or 'default' if the default context is unique.

class `URLParser.`**`ServletFactoryManagerClass`**

Bases: `object`

Manage servlet factories.

This singleton (called *ServletFactoryManager*) collects and manages all the servlet factories that are installed.

See *addServletFactory* for adding new factories, and *servletForFile* for getting the factories back.

__init__()

addServletFactory(*factory*)

Add a new servlet factory.

Servlet factories can add themselves with:

```
ServletFactoryManager.addServletFactory(factory)
```

The factory must have an *extensions* method, which should return a list of extensions that the factory handles (like `['.ht']`). The special extension `.*` will match any file if no other factory is found. See *ServletFactory* for more information.

factoryForFile(*path*)

Get a factory for a filename.

reset()

servletForFile(*trans*, *path*)

Get a servlet for a filename and transaction.

Uses *factoryForFile* to find the factory, which creates the servlet.

class URLParser.**URLParameterParser**(*fileParser=None*)

Bases: *URLParser*

Strips named parameters out of the URL.

E.g. in `/path/SID=123/etc` the `SID=123` will be removed from the URL, and a field will be set in the request (so long as no field by that name already exists – if a field does exist the variable is thrown away). These are put in the place of GET or POST variables.

It should be put in an __init__, like:

```
from URLParser import URLParameterParser
urlParserHook = URLParameterParser()
```

Or (slightly less efficient):

from URLParser import URLParameterParser as SubParser

__init__(*fileParser=None*)

findServletForTransaction(*trans*)

Returns a servlet for the transaction.

This is the top-level entry point, below it *parse* is used.

parse(*trans*, *requestPath*)

Delegates to *parseHook*.

static parseHook(*trans*, *requestPath*, *hook*)

Munges the path.

The *hook* is the FileParser object that originally called this – we just want to strip stuff out of the URL and then give it back to the FileParser instance, which can actually find the servlet.

`class URLParser.`**`URLParser`**

Bases: `object`

URLParser is the base class for all URL parsers.

Though its functionality is sparse, it may be expanded in the future. Subclasses should implement a *parse* method, and may also want to implement an *__init__* method with arguments that control how the parser works (for instance, passing a starting path for the parser)

The *parse* method is where most of the work is done. It takes two arguments – the transaction and the portion of the URL that is still to be parsed. The transaction may (and usually is) modified along the way. The URL is passed through so that you can take pieces off the front, and then pass the reduced URL to another parser. The method should return a servlet (never None).

If you cannot find a servlet, or some other (somewhat) expected error occurs, you should raise an exception. HTTPNotFound probably being the most interesting.

`findServletForTransaction``(`*`trans`*`)`

Returns a servlet for the transaction.

This is the top-level entry point, below it *parse* is used.

`URLParser.`**`application`**`()`

Returns the global Application.

`URLParser.`**`initApp`**`(`*`app`*`)`

Initialize the application.

Installs the proper servlet factories, and gets some settings from Application.config. Also saves the application in _globalApplication for future calls to the application() function.

This needs to be called before any of the URLParser-derived classes are instantiated.

`URLParser.`**`initParser`**`(`*`app`*`)`

Initialize the FileParser Class.

19.1.33 WSGIStreamOut

This module defines a class for writing responses using WSGI.

`exception WSGIStreamOut.`**`InvalidCommandSequence`**

Bases: `ConnectionError`

Invalid command sequence error

`__init__``(`*`*args`*`, `*`**kwargs`*`)`

`args`

`characters_written`

`errno`

POSIX exception code

`filename`

exception filename

`filename2`

second exception filename

strerror
: exception strerror

with_traceback()
: Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class WSGIStreamOut.**WSGIStreamOut**(*startResponse*, *autoCommit=False*, *bufferSize=8192*, *useWrite=True*, *encoding='utf-8'*)

Bases: `object`

This is a response stream to the client using WSGI.

The key attributes of this class are:

***_startResponse*:**
The start_response() function that is part of the WSGI protocol.

***_autoCommit*:**
If True, the stream will automatically start sending data once it has accumulated *_bufferSize* data. This means that it will ask the response to commit itself, without developer interaction. By default, this is set to False.

***_bufferSize*:**
The size of the data buffer. This is only used when autocommit is True. If not using autocommit, the whole response is buffered and sent in one shot when the servlet is done.

***_useWrite*:**
Whether the write callable that is returned by start_response() shall be used to deliver the response.

***flush()*:**
Send the accumulated response data now. Will ask the *Response* to commit if it hasn't already done so.

__init__(*startResponse*, *autoCommit=False*, *bufferSize=8192*, *useWrite=True*, *encoding='utf-8'*)

autoCommit()
Get the auto commit mode.

buffer()
Return accumulated data which has not yet been flushed.

We want to be able to get at this data without having to call flush() first, so that we can (for example) integrate automatic HTML validation.

bufferSize()
Get the buffer size.

clear()
Try to clear any accumulated response data.

Will fail if the response is already committed.

close()
Close this buffer. No more data may be sent.

closed()
Check whether we are closed to new data.

commit(*autoCommit=True*)
Called by the Response to tell us to go.

If *_autoCommit* is True, then we will be placed into autoCommit mode.

committed()
: Check whether the outptu is already committed

flush()
: Flush stream.

iterable()
: Return the WSGI iterable.

needCommit()
: Request for commitment.

  Called by the *HTTPResponse* instance that is using this instance to ask if the response needs to be prepared to be delivered. The response should then commit its headers, etc.

pop(*count*)
: Remove count bytes from the front of the buffer.

prepend(*output*)
: Add the output to the front of the response buffer.

  The output may be a byte string or anything that can be converted to a string and encoded to a byte string using the output encoding.

  Invalid if we are already committed.

setAutoCommit(*autoCommit=True*)
: Set the auto commit mode.

setBufferSize(*bufferSize=8192*)
: Set the buffer size.

size()
: Return the current size of the data held here.

startResponse(*status*, *headers*)
: Start the response with the given status and headers.

write(*output*)
: Write output to the buffer.

  The output may be a byte string or anything that can be converted to a string and encoded to a byte string using the output encoding.

19.1.34 XMLRPCServlet

XML-RPC servlet base class

Written by Geoffrey Talvola

See Examples/XMLRPCExample.py for sample usage.

class XMLRPCServlet.**XMLRPCServlet**
: Bases: *RPCServlet*

  XMLRPCServlet is a base class for XML-RPC servlets.

  See Examples/XMLRPCExample.py for sample usage.

  For more Pythonic convenience at the cost of language independence, see PickleRPCServlet.

`__init__()`

Subclasses must invoke super.

`allow_none = True`

`awake(`*transaction*`)`

Begin transaction.

`call(`*methodName*, **args*, ***keywords*`)`

Call custom method.

Subclasses may override this class for custom handling of methods.

`canBeReused()`

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

`canBeThreaded()`

Return whether the servlet can be multithreaded.

This value should not change during the lifetime of the object. The default implementation returns False. Note: This is not currently used.

`close()`

`exposedMethods()`

Get exposed methods.

Subclasses should return a list of methods that will be exposed through XML-RPC.

`static handleException(`*transaction*`)`

Handle exception.

If ReportRPCExceptionsInWebware is set to True, then flush the response (because we don't want the standard HTML traceback to be appended to the response) and then handle the exception in the standard Webware way. This means logging it to the console, storing it in the error log, sending error email, etc. depending on the settings.

`lastModified(`*_trans*`)`

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`log(`*message*`)`

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`name()`

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented(`*trans*`)`

`open()`

`respond(transaction)`

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, . . .).

`respondToHead(trans)`

Respond to a HEAD request.

A correct but inefficient implementation.

`respondToPost(transaction)`

Respond to a Post request.

This is similar to the xmlrpcserver.py example from the xmlrpc library distribution, only it’s been adapted to work within a Webware servlet.

`resultForException(e, trans)`

Get text for exception.

Given an unhandled exception, returns the string that should be sent back in the RPC response as controlled by the RPCExceptionReturn setting.

`runMethodForTransaction(transaction, method, *args, **kw)`

`static runTransaction(transaction)`

`static sendOK(contentType, contents, trans, contentEncoding=None)`

Send a 200 OK response with the given contents.

`serverSidePath(path=None)`

Return the filesystem path of the page on the server.

`setFactory(factory)`

`sleep(transaction)`

End transaction.

`transaction()`

Get the corresponding transaction.

Most uses of RPC will not need this.

19.2 PSP

19.2.1 BraceConverter

BraceConverter.py

Contributed 2000-09-04 by Dave Wallace (dwallace@delanet.com)

Converts Brace-blocked Python into normal indented Python. Brace-blocked Python is non-indentation aware and blocks are delimited by ‘:{’ and ‘}’ pairs.

Thus:

```
for x in range(10) :{
    if x % 2 :{ print(x) } else :{ print(z) }
}
```

Becomes (roughly, barring some spurious whitespace):

```
for x in range(10):
    if x % 2:
        print(x)
    else:
        print(z)
```

This implementation is fed a line at a time via parseLine(), outputs to a PSPServletWriter, and tracks the current quotation and block levels internally.

`class PSP.BraceConverter.`**`BraceConverter`**

Bases: `object`

`__init__``()`

`closeBrace``(`*writer*`)`

Close brace encountered.

`handleQuote``(`*quote*, *writer*`)`

Check and handle if current pos is a single or triple quote.

`openBlock``(`*writer*`)`

Open a new block.

`openBrace``(`*writer*`)`

Open brace encountered.

`parseLine``(`*line*, *writer*`)`

Parse a line.

The only public method of this class, call with subsequent lines and an instance of PSPServletWriter.

`skipQuote``(`*writer*`)`

Skip to end of quote.

Skip over all chars until the line is exhausted or the current non-escaped quote sequence is encountered.

19.2.2 Context

Utility class for keeping track of the context.

A utility class that holds information about the file we are parsing and the environment we are doing it in.

This software is based in part on work done by the Jakarta group.

class PSP.Context.**PSPCLContext**(*pspfile*)

Bases: *PSPContext*

A context for command line compilation.

Currently used for both command line and PSPServletEngine compilation. This class provides all the information necessary during the parsing and page generation steps of the PSP compilation process.

__init__(*pspfile*)

getBaseUri()

Return the base URI for the servlet.

getClassPath()

getFullClassName()

Return the class name including package prefixes.

Won't use this for now.

getFullPspFileName()

Return the name of the PSP file including its file path.

getOutputDirectory()

Provide directory to dump PSP source file to.

I am probably doing this in reverse order at the moment. I should start with this and get the Python filename from it.

getPspFileName()

Return the name of the PSP file from which we are generating.

getPythonFileEncoding()

Return the encoding of the file that we are generating.

getPythonFileName()

Return the filename that we are generating to.

getReader()

Return the PSPReader object assigned to this context.

getServletClassName()

Return the class name of the servlet being generated.

getServletWriter()

Return the ServletWriter object assigned to this context.

getWriter()

resolveRelativeURI(*uri*)

This is used mainly for including files.

It simply returns the location relative to the base context directory, ie Examples/. If the filename has a leading /, it is assumed to be an absolute path.

setClassName(*name*)

Set the class name to create.

setPSPReader(*reader*)

Set the PSPReader for this context.

setPythonFileEncoding(*encoding*)

Set the encoding of the .py file to generate.

setPythonFileName(*name*)

Set the name of the .py file to generate.

setServletWriter(*writer*)

Set the ServletWriter instance for this context.

class PSP.Context.**PSPContext**

Bases: `object`

PSPContext is an abstract base class for Context classes.

Holds all the common stuff that various parts of the compilation will need access to. The items in this class will be used by both the compiler and the class generator.

getClassPath()

getFullClassName()

Return the class name including package prefixes.

Won't use this for now.

getOutputDirectory()

Provide directory to dump PSP source file to.

getPythonFileEncoding()

Return the encoding of the file that we are generating.

getPythonFileName()

Return the filename that we are generating to.

getReader()

getServletClassName()

Return the class name of the servlet being generated.

getWriter()

setPSPReader(*reader*)

Set the PSPReader for this context.

setPythonFileEncoding(*encoding*)

Set the encoding of the .py file to generate.

setPythonFileName(*name*)

Set the name of the .py file to generate.

setServletWriter(*writer*)

Set the PSPWriter instance for this context.

19.2.3 Generators

Generate Python code from PSP templates.

This module holds the classes that generate the Python code resulting from the PSP template file. As the parser encounters PSP elements, it creates a new Generator object for that type of element. Each of these elements is put into a list maintained by the ParseEventHandler object. When it comes time to output the source code, each generator is called in turn to create its source.

This software is based in part on work done by the Jakarta group.

`class PSP.Generators.`**`CharDataGenerator`**`(`*chars*`)`

Bases: *`GenericGenerator`*

This class handles standard character output, mostly HTML.

It just dumps it out. Need to handle all the escaping of characters. It's just skipped for now.

`__init__``(`*chars*`)`

`generate``(`*writer*, *phase=None*`)`

`generateChunk``(`*writer*, *start=0*, *stop=None*`)`

`mergeData``(`*cdGen*`)`

`class PSP.Generators.`**`EndBlockGenerator`**

Bases: *`GenericGenerator`*

`__init__``()`

`generate``(`*writer*, *phase=None*`)`

`class PSP.Generators.`**`ExpressionGenerator`**`(`*chars*`)`

Bases: *`GenericGenerator`*

This class handles expression blocks.

It simply outputs the (hopefully) python expression within the block wrapped with a _formatter() call.

`__init__``(`*chars*`)`

`generate``(`*writer*, *phase=None*`)`

`class PSP.Generators.`**`GenericGenerator`**`(`*ctxt=None*`)`

Bases: `object`

Base class for all the generators

`__init__``(`*ctxt=None*`)`

class PSP.Generators.**IncludeGenerator**(*attrs*, *param*, *ctxt*)

Bases: *GenericGenerator*

Handle psp:include directives.

This is a new version of this directive that actually forwards the request to the specified page.

__init__(*attrs*, *param*, *ctxt*)

generate(*writer*, *phase=None*)

Just insert theFunction.

class PSP.Generators.**InsertGenerator**(*attrs*, *param*, *ctxt*)

Bases: *GenericGenerator*

Include files designated by the psp:insert syntax.

If the attribute 'static' is set to True or 1, we include the file now, at compile time. Otherwise, we use a function added to every PSP page named `__includeFile`, which reads the file at run time.

__init__(*attrs*, *param*, *ctxt*)

generate(*writer*, *phase=None*)

class PSP.Generators.**MethodEndGenerator**(*chars*, *attrs*)

Bases: *GenericGenerator*

Part of class method generation.

After MethodGenerator, MethodEndGenerator actually generates the code for the method body.

__init__(*chars*, *attrs*)

generate(*writer*, *phase=None*)

class PSP.Generators.**MethodGenerator**(*chars*, *attrs*)

Bases: *GenericGenerator*

Generate class methods defined in the PSP page.

There are two parts to method generation. This class handles getting the method name and parameters set up.

__init__(*chars*, *attrs*)

generate(*writer*, *phase=None*)

class PSP.Generators.**ScriptClassGenerator**(*chars*, *attrs*)

Bases: *GenericGenerator*

Add Python code at the class level.

__init__(*chars*, *attrs*)

generate(*writer*, *phase=None*)

class PSP.Generators.**ScriptFileGenerator**(*chars*, *attrs*)

Bases: *GenericGenerator*

Add Python code at the file/module level.

__init__(*chars*, *attrs*)

generate(*writer*, *phase=None*)

class PSP.Generators.**ScriptGenerator**(*chars*, *attrs*)

Bases: `GenericGenerator`

Generate scripts.

__init__(*chars*, *attrs*)

generate(*writer*, *phase=None*)

19.2.4 ParseEventHandler

Event handler for parsing PSP tokens.

This module is called when the Parser encounters psp tokens. It creates a generator to handle the PSP token. When the PSP source file is fully parsed, this module calls all of the generators in turn to output their source code.

Copyright (c) by Jay Love, 2000 (mailto:jsliv@jslove.org)

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee or royalty is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation or portions thereof, including modifications, that you make.

This software is based in part on work done by the Jakarta group.

class PSP.ParseEventHandler.**ParseEventHandler**(*ctxt*, *parser*)

Bases: `object`

This is a key class.

It implements the handling of all the parsing elements. Note: This files JSP cousin is called ParseEventListener, I don't know why, but Handler seemed more appropriate to me.

__init__(*ctxt*, *parser*)

addGenerator(*gen*)

aspace = ' '

beginProcessing()

defaults = {'BASE_CLASS': 'Page', 'BASE_METHOD': 'writeHTML', 'formatter': 'str', 'gobbleWhitespace': True, 'imports': {'filename': 'classes'}, 'indent': 4, 'instanceSafe': 'yes', 'threadSafe': 'no'}

directiveHandlers = {'BaseClass': <function ParseEventHandler.extendsHandler>, 'extends': <function ParseEventHandler.extendsHandler>, 'formatter': <function ParseEventHandler.formatterHandler>, 'gobbleWhitespace': <function ParseEventHandler.gobbleWhitespaceHandler>, 'import': <function ParseEventHandler.importHandler>, 'imports': <function ParseEventHandler.importHandler>, 'indentSpaces': <function ParseEventHandler.indentSpacesHandler>, 'indentType': <function ParseEventHandler.indentTypeHandler>, 'isInstanceSafe': <function ParseEventHandler.instanceSafeHandler>, 'isThreadSafe': <function ParseEventHandler.threadSafeHandler>, 'method': <function ParseEventHandler.mainMethodHandler>}

endProcessing()

extendsHandler(*bases*, *start*, *stop*)

Extends is a page directive.

It sets the base class (or multiple base classes) for the class that this class will generate. The choice of base class affects the choice of a method to override with the BaseMethod page directive. The default base class is PSPPage. PSPPage inherits from Page.py.

formatterHandler(*value*, *start*, *stop*)

Set an alternate formatter function to use instead of str().

generateAll(*phase*)

generateDeclarations()

generateFooter()

generateHeader()

generateInitPSP()

generateMainMethod()

gobbleWhitespace()

Gobble up whitespace.

This method looks for a character block between two PSP blocks that contains only whitespace. If it finds one, it deletes it.

This is necessary so that a write() line can't sneek in between a if/else, try/except etc.

gobbleWhitespaceHandler(*value*, *start*, *stop*)

Declare whether whitespace between script tags are gobble up.

handleCharData(*start*, *stop*, *chars*)

Flush character data into a CharDataGenerator.

handleComment(*start*, *stop*)

Comments get swallowed into nothing.

handleDirective(*directive*, *start*, *stop*, *attrs*)

Flush any template data and create a new DirectiveGenerator.

handleEndBlock()

handleExpression(*start*, *stop*, *attrs*)

Flush any template data and create a new ExpressionGenerator.

handleInclude(*attrs*, *param*)

This is for includes of the form <psp:include . . . >

This function essentially forwards the request to the specified URL and includes that output.

handleInsert(*attrs*, *param*)

This is for includes of the form <psp:insert . . . >

This type of include is not parsed, it is just inserted into the output stream.

handleMethod(*start*, *stop*, *attrs*) `handleMethodEnd`(*start*, *stop*, *attrs*)

`handleScript`(*start*, *stop*, *attrs*)
Handle scripting elements

`handleScriptClass`(*start*, *stop*, *attrs*)
Python script that goes at the class level

`handleScriptFile`(*start*, *stop*, *attrs*)
Python script that goes at the file/module level

`importHandler`(*imports*, *start*, *stop*)

`indentSpacesHandler`(*amount*, *start*, *stop*)
Set number of spaces used to indent in generated source.

`indentTypeHandler`(*indentType*, *start*, *stop*)
Declare whether tabs are used to indent source code.

`instanceSafeHandler`(*value*, *start*, *stop*)
Handle isInstanceSafe.

isInstanceSafe tells the Servlet engine whether it is safe to use object instances of this page multiple times. The default is “yes”.

Saying “no” here hurts performance.

`mainMethodHandler`(*method*, *start*, *stop*)
BaseMethod is a page directive.

It sets the class method that the main body of this PSP page over-rides. The default is WriteHTML. This value should be set to either WriteHTML or writeBody. See the PSPPage.py and Page.py servlet classes for more information.

`optimizeCharData`()
Optimize the CharData.

Too many char data generators make the servlet slow. If the current Generator and the next are both CharData type, merge their data.

`setTemplateInfo`(*start*, *stop*)
Mark non code data.

`threadSafeHandler`(*value*, *start*, *stop*)
Handle isThreadSage.

isThreadSafe is a page directive. The value can be “yes” or “no”. Default is no because the default base class, Page.py, isn’t thread safe.

`PSP.ParseEventHandler.checkForTextHavingOnlyGivenChars`(*text*, *whitespace=None*)
Checks whether text contains only whitespace (or other chars).

Does the given text contain anything other than the whitespace characters? Return true if text is only whitespace characters.

19.2.5 PSPCompiler

A simple little module that organizes the actual page generation.

Copyright (c) by Jay Love, 2000 (mailto:jsliv@jslove.org)

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee or royalty is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation or portions thereof, including modifications, that you make.

This software is based in part on work done by the Jakarta group.

`class PSP.PSPCompiler.Compiler(`*context*`)`

Bases: `object`

The main compilation class.

`__init__(`*context*`)`

`compile()`

Compile the PSP context and return a set of all source files.

19.2.6 PSPPage

Default base class for PSP pages.

This class is intended to be used in the future as the default base class for PSP pages in the event that some special processing is needed. Right now, no special processing is needed, so the default base class for PSP pages is the standard Webware Page.

`class PSP.PSPPage.PSPPage`

Bases: *Page*

`__init__()`

Subclasses must invoke super.

`actions()`

The allowed actions.

Returns a list or a set of method names that are allowable actions from HTML forms. The default implementation returns []. See *_respond* for more about actions.

`application()`

The *Application* instance we're using.

`awake(`*transaction*`)`

Let servlet awake.

Makes instance variables from the transaction. This is where Page becomes unthreadsafe, as the page is tied to the transaction. This is also what allows us to implement functions like *write*, where you don't need to pass in the transaction or response.

`callMethodOfServlet(`*url*, *method*, **args*, ***kwargs*`)`

Call a method of another servlet.

See *Application.callMethodOfServlet* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`canBeReused()`

Returns whether a single servlet instance can be reused.

The default is True, but subclasses con override to return False. Keep in mind that performance may seriously be degraded if instances can't be reused. Also, there's no known good reasons not to reuse an instance. Remember the awake() and sleep() methods are invoked for every transaction. But just in case, your servlet can refuse to be reused.

`canBeThreaded()`

Declares whether servlet can be threaded.

Returns False because of the instance variables we set up in *awake*.

`close()`

`defaultAction()`

The default action in a Page is to writeHTML().

`static endResponse()`

End response.

When this method is called during *awake* or *respond*, servlet processing will end immediately, and the accumulated response will be sent.

Note that *sleep* will still be called, providing a chance to clean up or free any resources.

`forward(`*url*`)`

Forward request.

Forwards this request to another servlet. See *Application.forward* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`handleAction(`*action*`)`

Handle action.

Invoked by *_respond* when a legitimate action has been found in a form. Invokes *preAction*, the actual action method and *postAction*.

Subclasses rarely override this method.

`htBodyArgs()`

The attributes for the <body> element.

Returns the arguments used for the HTML <body> tag. Invoked by writeBody().

With the prevalence of stylesheets (CSS), you can probably skip this particular HTML feature, but for historical reasons this sets the page to black text on white.

`htRootArgs()`

The attributes for the <html> element.

Returns the arguments used for the root HTML tag. Invoked by writeHTML() and preAction().

Authors are encouraged to specify a lang attribute, giving the document's language.

`htTitle()`

The page title as HTML.

Return self.title(). Subclasses sometimes override this to provide an HTML enhanced version of the title. This is the method that should be used when including the page title in the actual page contents.

`static htmlDecode(s)`

HTML decode special characters.

Alias for `WebUtils.Funcs.htmlDecode`. Decodes HTML entities.

`static htmlEncode(s)`

HTML encode special characters. Alias for `WebUtils.Funcs.htmlEncode`, quotes the special characters &, <, >, and "

`includeURL(url)`

Include output from other servlet.

Includes the response of another servlet in the current servlet's response. See *Application.includeURL* for details. The main difference is that here you don't have to pass in the transaction as the first argument.

`lastModified(_trans)`

Get time of last modification.

Return this object's Last-Modified time (as a float), or None (meaning don't know or not applicable).

`log(message)`

Log a message.

This can be invoked to print messages concerning the servlet. This is often used by self to relay important information back to developers.

`methodNameForAction(name)`

Return method name for an action name.

Invoked by _respond() to determine the method name for a given action name which has been derived as the value of an `_action_` field. Since this is usually the label of an HTML submit button in a form, it is often needed to transform it in order to get a valid method name (for instance, blanks could be replaced by underscores and the like). This default implementation of the name transformation is the identity, it simply returns the name. Subclasses should override this method when action names don't match their method names; they could "mangle" the action names or look the method names up in a dictionary.

`name()`

Return the name which is simple the name of the class.

Subclasses should *not* override this method. It is used for logging and debugging.

`static notImplemented(trans)`

`open()`

`outputEncoding()`

Get the default output encoding of the application.

`postAction(actionName)`

Things to do after actions.

Simply close the html tag (</html>).

`preAction(actionName)`

Things to do before actions.

For a page, we first writeDocType(), <html>, and then writeHead().

`request()`

The request (*HTTPRequest*) we're handling.

`respond`(*transaction*)

Respond to a request.

Invokes the appropriate respondToSomething() method depending on the type of request (e.g., GET, POST, PUT, . . .).

`respondToGet`(*transaction*)

Respond to GET.

Invoked in response to a GET request method. All methods are passed to *_respond*.

`respondToHead`(*trans*)

Respond to a HEAD request.

A correct but inefficient implementation.

`respondToPost`(*transaction*)

Respond to POST.

Invoked in response to a POST request method. All methods are passed to *_respond*.

`response`()

The response (*HTTPResponse*) we're handling.

`runMethodForTransaction`(*transaction*, *method*, **args*, ***kw*)

`static runTransaction`(*transaction*)

`sendRedirectAndEnd`(*url*, *status=None*)

Send redirect and end.

Sends a redirect back to the client and ends the response. This is a very popular pattern.

`sendRedirectPermanentAndEnd`(*url*)

Send permanent redirect and end.

`sendRedirectSeeOtherAndEnd`(*url*)

Send redirect to a URL to be retrieved with GET and end.

This is the proper method for the Post/Redirect/Get pattern.

`sendRedirectTemporaryAndEnd`(*url*)

Send temporary redirect and end.

`serverSidePath`(*path=None*)

Return the filesystem path of the page on the server.

`session`()

The session object.

This provides a state for the current user (associated with a browser instance, really). If no session exists, then a session will be created.

`sessionEncode`(*url=None*)

Utility function to access *Session.sessionEncode*.

Takes a url and adds the session ID as a parameter. This is for cases where you don't know if the client will accepts cookies.

`setFactory`(*factory*)

`sleep`(*transaction*)

Let servlet sleep again.

We unset some variables. Very boring.

`title`()

The page title.

Subclasses often override this method to provide a custom title. This title should be absent of HTML tags. This implementation returns the name of the class, which is sometimes appropriate and at least informative.

`transaction`()

The *Transaction* we're currently handling.

static `urlDecode`(*s*)

Turn special % characters into actual characters.

This method does the same as the *urllib.unquote_plus()* function.

static `urlEncode`(*s*)

Quotes special characters using the % substitutions.

This method does the same as the *urllib.quote_plus()* function.

`write`(**args*)

Write to output.

Writes the arguments, which are turned to strings (with *str*) and concatenated before being written to the response. Unicode strings must be encoded before they can be written.

`writeBody`()

Write the <body> element of the page.

Writes the <body> portion of the page by writing the `<body>...</body>` (making use of `htBodyArgs`) and invoking `writeBodyParts` in between.

`writeBodyParts`()

Write the parts included in the <body> element.

Invokes `writeContent`. Subclasses should only override this method to provide additional page parts such as a header, sidebar and footer, that a subclass doesn't normally have to worry about writing.

For writing page-specific content, subclasses should override `writeContent` instead. This method is intended to be overridden by your SitePage.

See `SidebarPage` for an example override of this method.

Invoked by `writeBody`.

`writeContent`()

Write the unique, central content for the page.

Subclasses should override this method (not invoking super) to write their unique page content.

Invoked by `writeBodyParts`.

`writeDocType`()

Write the DOCTYPE tag.

Invoked by `writeHTML` to write the `<!DOCTYPE ...>` tag.

By default this gives the HTML 5 DOCTYPE.

Subclasses may override to specify something else.

writeExceptionReport(*handler*)

Write extra information to the exception report.

The *handler* argument is the exception handler, and information is written there (using *writeTitle*, *write*, and *writeln*). This information is added to the exception report.

See *ExceptionHandler* for more information.

writeHTML()

Write all the HTML for the page.

Subclasses may override this method (which is invoked by `_respond`) or more commonly its constituent methods, `writeDocType`, `writeHead` and `writeBody`.

You will want to override this method if:

- you want to format the entire HTML page yourself
- if you want to send an HTML page that has already been generated
- if you want to use a template that generates the entire page
- if you want to send non-HTML content; in this case, be sure to call self.response().setHeader('Content-Type', 'mime/type').

writeHead()

Write the <head> element of the page.

Writes the `<head>` portion of the page by writing the `<head>...</head>` tags and invoking `writeHeadParts` in between.

writeHeadParts()

Write the parts included in the <head> element.

Writes the parts inside the `<head>...</head>` tags. Invokes `writeTitle` and then `writeMetaData`, `writeStyleSheet` and `writeJavaScript`. Subclasses should override the `title` method and the three latter methods only.

writeJavaScript()

Write the JavaScript for the page.

This default implementation does nothing. Subclasses should override if necessary.

A typical implementation is:

```
self.writeln('<script src="ajax.js"></script>')
```

writeMetaData()

Write the meta data for the page.

This default implementation only specifies the output encoding. Subclasses should override if necessary.

writeStyleSheet()

Write the CSS for the page.

This default implementation does nothing. Subclasses should override if necessary.

A typical implementation is:

```
self.writeln('<link rel="stylesheet" href="StyleSheet.css">')
```

writeTitle()

Write the <title> element of the page.

Writes the <`title`> portion of the page. Uses `title`, which is where you should override.

writeln(**args*)

Write to output with newline.

Writes the arguments (like *write*), adding a newline after. Unicode strings must be encoded before they can be written.

19.2.7 PSPParser

The PSP parser.

This module handles the actual reading of the characters in the source PSP file and checking it for valid psp tokens. When it finds one, it calls ParseEventHandler with the characters it found.

Copyright (c) by Jay Love, 2000 (mailto:jsliv@jslove.org)

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee or royalty is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation or portions thereof, including modifications, that you make.

This software is based in part on work done by the Jakarta group.

class PSP.PSPParser.**PSPParser**(*ctxt*)

Bases: `object`

The main PSP parser class.

The PSPParser class does the actual sniffing through the input file looking for anything we're interested in. Basically, it starts by looking at the code looking for a '<' symbol. It looks at the code by working with a PSPReader object, which handles the current location in the code. When it finds one, it calls a list of checker methods, asking each if it recognizes the characters as its kind of input. When the checker methods look at the characters, if they want it, they go ahead and gobble it up and set up to create it in the servlet when the time comes. When they return, they return true if they accept the character, and the PSPReader object cursor is positioned past the end of the block that the checker method accepted.

__init__(*ctxt*)

checkDirective(*handler*, *reader*)

Check for directives; for now we support only page and include.

checkEndBlock(*handler*, *reader*)

Check for the end of a block.

checkExpression(*handler*, *reader*)

Look for "expressions" and handle them.

checkInclude(*handler*, *reader*)

Check for inserting another pages output in this spot.

checkInsert(*handler*, *reader*)

Check for straight character dumps.

No big hurry for this. It's almost the same as the page include directive. This is only a partial implementation of what JSP does. JSP can pull it from another server, servlet, JSP page, etc.

checkMethod(*handler*, *reader*)

Check for class methods defined in the page.

We only support one format for these, `<psp:method name="xxx" params="xxx,xxx">` Then the function body, then </psp:method>.

checkScript(*handler*, *reader*)

The main thing we're after. Check for embedded scripts.

checkScriptClass(*handler*, *reader*)

Check for class level code.

Check for Python code that should go in the class definition:

```
<psp:class>
    def foo(self):
        return self.dosomething()
</psp:class>
```

checkScriptFile(*handler*, *reader*)

Check for file level code.

Check for Python code that must go to the top of the generated module:

```
<psp:file>
    import xyz
    print('hi Mome!')
    def foo(): return 'foo'
</psp:file>
```

checklist = [<function PSPParser.commentCheck>, <function PSPParser.checkExpression>, <function PSPParser.checkDirective>, <function PSPParser.checkEndBlock>, <function PSPParser.checkScript>, <function PSPParser.checkScriptFile>, <function PSPParser.checkScriptClass>, <function PSPParser.checkMethod>, <function PSPParser.checkInclude>, <function PSPParser.checkInsert>]

commentCheck(*_handler*, *reader*)

Comments just get eaten.

flushCharData(*start*, *stop*)

Dump everything to the char data handler.

Dump all the HTML that we've accumulated over to the character data handler in the event handler object.

parse(*until=None*)

Parse the PSP file.

setEventHandler(*handler*)

Set the handler this parser will use when it finds PSP code.

PSP.PSPParser.**checker**(*method*)

Decorator for adding a method to the checklist.

19.2.8 PSPServletFactory

This module handles requests from the application for PSP pages.

`class PSP.PSPServletFactory.`**`PSPServletFactory`**`(`*application*`)`

Bases: *`ServletFactory`*

Servlet Factory for PSP files.

`__init__``(`*application*`)`

Create servlet factory.

Stores a reference to the application in self._app, because subclasses may or may not need to talk back to the application to do their work.

`clearFileCache``()`

Clear class files stored on disk.

`computeClassName``(`*pageName*`)`

Generates a (hopefully) unique class/file name for each PSP file.

Argument: pageName: the path to the PSP source file Returns: a unique name for the class generated fom this PSP source file

`extensions``()`

Return a list of extensions that match this handler.

Extensions should include the dot. An empty string indicates a file with no extension and is a valid value. The extension '.*' is a special case that is looked for a URL's extension doesn't match anything.

`fileEncoding``()`

Return the file encoding used in PSP files.

`flushCache``()`

Clean out the cache of classes in memory and on disk.

`importAsPackage``(`*transaction*, *serverSidePathToImport*`)`

Import requested module.

Imports the module at the given path in the proper package/subpackage for the current request. For example, if the transaction has the URL http://localhost/Webware/MyContextDirectory/MySubdirectory/MyPage and path = 'some/random/path/MyModule.py' and the context is configured to have the name 'MyContext' then this function imports the module at that path as MyContext.MySubdirectory.MyModule . Note that the context name may differ from the name of the directory containing the context, even though they are usually the same by convention.

Note that the module imported may have a different name from the servlet name specified in the URL. This is used in PSP.

`loadClass``(`*transaction*, *path*`)`

Load the appropriate class.

Given a transaction and a path, load the class for creating these servlets. Caching, pooling, and threadsafeness are all handled by servletForTransaction. This method is not expected to be threadsafe.

loadClassFromFile(*transaction*, *fileName*, *className*)

Create an actual class instance.

The module containing the class is imported as though it were a module within the context's package (and appropriate subpackages).

name()

Return the name of the factory.

This is a convenience for the class name.

returnServlet(*servlet*)

Return servlet to the pool.

Called by Servlet.close(), which returns the servlet to the servlet pool if necessary.

servletForTransaction(*transaction*)

Return a new servlet that will handle the transaction.

This method handles caching, and will call loadClass(trans, filepath) if no cache is found. Caching is generally controlled by servlets with the canBeReused() and canBeThreaded() methods.

uniqueness()

Return uniqueness type.

Returns a string to indicate the uniqueness of the ServletFactory's servlets. The Application needs to know if the servlets are unique per file, per extension or per application.

Return values are 'file', 'extension' and 'application'.

NOTE: Application so far only supports 'file' uniqueness.

19.2.9 PSPUtils

A bunch of utility functions for the PSP generator.

This software is based in part on work done by the Jakarta group.

exception PSP.PSPUtils.**PSPParserException**

Bases: `Exception`

PSP parser error.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

PSP.PSPUtils.**checkAttributes**(*tagType*, *attrs*, *validAttrs*)

Check for mandatory and optional atributes.

PSP.PSPUtils.**getExpr**(*s*)

Get the content of a PSP expression.

PSP.PSPUtils.**isExpression**(*s*)

Check whether this is a PSP expression.

PSP.PSPUtils.**normalizeIndentation**(*pySource*)

Take code block that may be too indented and move it all to the left.

PSP.PSPUtils.**removeQuotes**(*s*)

PSP.PSPUtils.**splitLines**(*text*, *keepends=False*)

Split text into lines.

PSP.PSPUtils.**startsNewBlock**(*line*)

Determine whether a code line starts a new block.

19.2.10 ServletWriter

This module holds the actual file writer class.

Copyright (c) by Jay Love, 2000 (mailto:jsliv@jslove.org)

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee or royalty is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation or portions thereof, including modifications, that you make.

This software is based in part on work done by the Jakarta group.

class PSP.ServletWriter.**ServletWriter**(*ctxt*)

Bases: object

This file creates the servlet source code.

Well, it writes it out to a file at least.

__init__(*ctxt*)

close()

indent(*s*)

Indent the string.

popIndent()

printChars(*s*)

Just prints what its given.

printComment(*start*, *stop*, *chars*)

printIndent()

Just prints tabs.

printList(*strList*)

Prints a list of strings with indentation and a newline.

printMultiLn(*s*)

println(*line=None*)

Print with indentation and a newline if none supplied.

pushIndent()

this is very key, have to think more about it

quoteString(*s*)

Escape the string.

setIndentSpaces(*amt*)

setIndentType(*indentType*)

setIndention()

19.2.11 StreamReader

This module co-ordinates the reading of the source file.

It maintains the current position of the parser in the source file.

Copyright (c) by Jay Love, 2000 (mailto:jsliv@jslove.org)

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee or royalty is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation or portions thereof, including modifications, that you make.

This software is based in part on work done by the Jakarta group.

class PSP.StreamReader.**Mark**(*reader*, *fileId=None*, *stream=None*, *inBaseDir=None*, *encoding=None*)

Bases: `object`

This class marks a point in an input stream.

__init__(*reader*, *fileId=None*, *stream=None*, *inBaseDir=None*, *encoding=None*)

getFile()

popStream()

pushStream(*inFileId*, *inStream*, *inBaseDir*, *inEncoding*)

class PSP.StreamReader.**StreamReader**(*filename*, *ctxt*)

Bases: `object`

This class handles the PSP source file.

It provides the characters to the other parts of the system. It can move forward and backwards in a file and remember locations.

__init__(*filename*, *ctxt*)

advance(*length*)

Advance length characters

getChars(*start*, *stop*)

getFile(*i*)

hasMoreInput()

init()

isDelimiter()

isSpace()

No advancing.

mark()

matches(*s*)

newSourceFile(*filename*)

nextChar()

nextContent()

Find next < char.

parseAttributeValue(*valueDict*)

parseTagAttributes()

Parse the attributes at the beginning of a tag.

parseToken(*quoted*)

peekChar(*cnt=1*)

popFile()

pushFile(*filepath*, *encoding=None*)

registerSourceFile(*filepath*)

reset(*mark*)

skipSpaces()

skipUntil(*s*)

Greedy search.

Return the point before the string, but move reader past it.

19.3 UserKit

19.3.1 HierRole

The HierRole class.

class UserKit.HierRole.**HierRole**(*name*, *description=None*, *superRoles=None*)

Bases: `Role`

HierRole is a hierarchical role.

It points to its parent roles. The hierarchy cannot have cycles.

__init__(*name*, *description=None*, *superRoles=None*) `description()`

`name()`

`playsRole(role)`

Check whether the receiving role plays the role that is passed in.

This implementation provides for the inheritance supported by HierRole.

`setDescription(description)`

`setName(name)`

19.3.2 Role

The basic Role class.

`class UserKit.Role.Role(name, description=None)`

Bases: `object`

Used in conjunction with RoleUser to provide role-based security.

All roles have a name and a description and respond to playsRole().

RoleUser also responds to playsRole() and is the more popular entry point for programmers. Application code may then do something along the lines of:

if user.playsRole('admin'):
self.displayAdminMenuItems()

See also:

- class HierRole
- class RoleUser

`__init__(name, description=None)`

`description()`

`name()`

`playsRole(role)`

Return true if the receiving role plays the role passed in.

For Role, this is simply a test of equality. Subclasses may override this method to provide richer semantics (such as hierarchical roles).

`setDescription(description)`

`setName(name)`

19.3.3 RoleUser

The RoleUser class.

class `UserKit.RoleUser.`**`RoleUser`**(*manager=None*, *name=None*, *password=None*)

Bases: `User`

In conjunction with Role, provides role-based users and security.

See the doc for playsRole() for an example.

Note that this class plays nicely with both Role and HierRole, e.g., no “HierRoleUser” is needed when making use of HierRoles.

See also:

- class Role
- class HierRole

`__init__`(*manager=None*, *name=None*, *password=None*)

`addRoles`(*listOfRoles*)

Add additional roles for the user.

Each role in the list may be a valid role name or a Role object.

`creationTime`()

`externalId`()

`isActive`()

`lastAccessTime`()

`lastLoginTime`()

`login`(*password*, *fromMgr=0*)

Return self if the login is successful and None otherwise.

`logout`(*fromMgr=False*)

`manager`()

`name`()

`password`()

`playsRole`(*roleOrName*)

Check whether the user plays the given role.

More specifically, if any of the user’s roles return true for role.playsRole(otherRole), this method returns True.

The application of this popular method often looks like this:

if user.playsRole(‘admin’):
self.displayAdminMenuItems()

`roles`()

Return a direct list of the user’s roles.

Do not modify.

serialNum()

setManager(*manager*)

Set the manager, which can only be done once.

setName(*name*)

Set the name, which can only be done once.

setPassword(*password*)

setRoles(*listOfRoles*)

Set all the roles for the user.

Each role in the list may be a valid role name or a Role object.

Implementation note: depends on addRoles().

setSerialNum(*serialNum*)

wasAccessed()

19.3.4 RoleUserManager

The RoleUserManager class.

class UserKit.RoleUserManager.**RoleUserManager**(*userClass=None*)

Bases: *UserManager*, *RoleUserManagerMixIn*

See the base classes for more information.

__init__(*userClass=None*)

activeUserTimeout()

activeUsers()

Return a list of all active users.

addRole(*role*)

addUser(*user*)

cachedUserTimeout()

clearCache()

Clear the cache of the manager.

Use with extreme caution. If your program maintains a reference to a user object, but the manager loads in a new copy later on, then consistency problems could occur.

The most popular use of this method is in the regression test suite.

clearRoles()

createUser(*name*, *password*, *userClass=None*)

Return a newly created user that is added to the manager.

If userClass is not specified, the manager's default user class is instantiated. This not imply that the user is logged in. This method invokes self.addUser().

See also: userClass(), setUserClass()

delRole(*name*)

hasRole(*name*)

inactiveUsers()

initUserClass()

Invoked by __init__ to set the default user class to RoleUser.

login(*user*, *password*)

Return the user if login is successful, otherwise return None.

loginExternalId(*externalId*, *password*)

loginName(*userName*, *password*)

loginSerialNum(*serialNum*, *password*)

logout(*user*)

modifiedUserTimeout()

numActiveUsers()

Return the number of active users (e.g. the logged in users).

role(*name*, *default=<class 'MiscUtils.NoDefault'>*)

roles()

setActiveUserTimeout(*value*)

setCachedUserTimeout(*value*)

setModifiedUserTimeout(*value*)

setUserClass(*userClass*)

Set the userClass, which cannot be None and must inherit from User.

See also: userClass().

shutDown()

Perform any tasks necessary to shut down the user manager.

Subclasses may override and must invoke super as their *last* step.

userClass()

Return the userClass, which is used by createUser.

The default value is UserKit.User.User.

userForExternalId(*externalId*, *default=<class 'MiscUtils.NoDefault'>*)

Return the user with the given external id.

The user record is pulled into memory if needed.

userForName(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return the user with the given name.

The user record is pulled into memory if needed.

userForSerialNum(*serialNum*, *default=<class 'MiscUtils.NoDefault'>*)

Return the user with the given serialNum.

The user record is pulled into memory if needed.

users()

Return a list of all users (regardless of login status).

19.3.5 RoleUserManagerMixIn

The RoleUserManager mixin.

class UserKit.RoleUserManagerMixIn.**RoleUserManagerMixIn**

Bases: `object`

Mixin class for mapping names to roles.

This mixin adds the functionality of keeping a dictionary mapping names to role instances. Several accessor methods are provided for this.

__init__()

addRole(*role*)

clearRoles()

delRole(*name*)

hasRole(*name*)

initUserClass()

Invoked by __init__ to set the default user class to RoleUser.

role(*name*, *default=<class 'MiscUtils.NoDefault'>*)

roles()

19.3.6 RoleUserManagerToFile

The RoleUserManagerToFile class.

class UserKit.RoleUserManagerToFile.**RoleUserManagerToFile**(*userClass=None*)

Bases: *RoleUserManagerMixIn*, *UserManagerToFile*

See the base classes for more information.

__init__(*userClass=None*)

activeUserTimeout()

activeUsers()

Return a list of all active users.

addRole(*role*)

addUser(*user*)

cachedUserTimeout()

clearCache()

Clear the cache of the manager.

Use with extreme caution. If your program maintains a reference to a user object, but the manager loads in a new copy later on, then consistency problems could occur.

The most popular use of this method is in the regression test suite.

clearRoles()

createUser(*name*, *password*, *userClass=None*)

Return a newly created user that is added to the manager.

If userClass is not specified, the manager's default user class is instantiated. This not imply that the user is logged in. This method invokes self.addUser().

See also: userClass(), setUserClass()

decoder()

delRole(*name*)

encoder()

hasRole(*name*)

inactiveUsers()

initNextSerialNum()

initUserClass()

Invoked by __init__ to set the default user class to RoleUser.

loadUser(*serialNum*, *default=<class 'MiscUtils.NoDefault'>*)

Load the user with the given serial number from disk.

If there is no such user, a KeyError will be raised unless a default value was passed, in which case that value is returned.

login(*user*, *password*)

Return the user if login is successful, otherwise return None.

loginExternalId(*externalId*, *password*)

loginName(*userName*, *password*)

loginSerialNum(*serialNum*, *password*)

logout(*user*)

modifiedUserTimeout()

nextSerialNum()

numActiveUsers()

Return the number of active users (e.g. the logged in users).

role(*name*, *default=<class 'MiscUtils.NoDefault'>*)

roles()

scanSerialNums()
: Return a list of all the serial numbers of users found on disk.

  Serial numbers are always integers.

setActiveUserTimeout(*value*)

setCachedUserTimeout(*value*)

setEncoderDecoder(*encoder*, *decoder*)

setModifiedUserTimeout(*value*)

setUserClass(*userClass*)
: Overridden to mix in UserMixIn to the class that is passed in.

setUserDir(*userDir*)
: Set the directory where user information is stored.

  You should strongly consider invoking initNextSerialNum() afterwards.

shutDown()
: Perform any tasks necessary to shut down the user manager.

  Subclasses may override and must invoke super as their *last* step.

userClass()
: Return the userClass, which is used by createUser.

  The default value is UserKit.User.User.

userDir()

userForExternalId(*externalId*, *default=<class 'MiscUtils.NoDefault'>*)
: Return the user with the given external id.

  The user record is pulled into memory if needed.

userForName(*name*, *default=<class 'MiscUtils.NoDefault'>*)
: Return the user with the given name.

  The user record is pulled into memory if needed.

userForSerialNum(*serialNum*, *default=<class 'MiscUtils.NoDefault'>*)
: Return the user with the given serialNum.

  The user record is pulled into memory if needed.

users()
: Return a list of all users (regardless of login status).

19.3.7 User

The basic User class.

class `UserKit.User.`**User**(*manager=None*, *name=None*, *password=None*)

Bases: `object`

The base class for a UserKit User.

`__init__`(*manager=None*, *name=None*, *password=None*)

`creationTime`()

`externalId`()

`isActive`()

`lastAccessTime`()

`lastLoginTime`()

`login`(*password*, *fromMgr=0*)

Return self if the login is successful and None otherwise.

`logout`(*fromMgr=False*)

`manager`()

`name`()

`password`()

`serialNum`()

`setManager`(*manager*)

Set the manager, which can only be done once.

`setName`(*name*)

Set the name, which can only be done once.

`setPassword`(*password*)

`setSerialNum`(*serialNum*)

`wasAccessed`()

19.3.8 UserManager

The abstract UserManager class.

class `UserKit.UserManager.`**UserManager**(*userClass=None*)

Bases: `object`

The base class for all user manager classes.

A UserManager manages a set of users including authentication, indexing and persistence. Keep in mind that UserManager is abstract; you must always use a concrete subclasses like UserManagerToFile (but please read the rest of this docstring).

You can create a user through the manager (preferred):

```
user = manager.createUser(name, password)
```

Or directly through the user class:

```
user = RoleUser(manager, name, password)
manager.addUser(user)
```

The manager tracks users by whether or not they are "active" (e.g., logged in) and indexes them by:

- user serial number
- external user id
- user name

These methods provide access to the users by these keys:

```
def userForSerialNum(self, serialNum, default=NoDefault)
def userForExternalId(self, extId, default=NoDefault)
def userForName(self, name, default=NoDefault)
```

UserManager provides convenient methods for iterating through the various users. Each method returns an object that can be used in a for loop and asked for its len():

```
def users(self)
def activeUsers(self)
def inactiveUsers(self)
```

You can authenticate a user by passing the user object and attempted password to login(). If the authentication is successful, then login() returns the User, otherwise it returns None:

```
user = mgr.userForExternalId(externalId)
if mgr.login(user, password):
    self.doSomething()
```

As a convenience, you can authenticate by passing the serialNum, externalId or name of the user:

```
def loginSerialNum(self, serialNum, password):
def loginExternalId(self, externalId, password):
def loginName(self, userName, password):
```

The user will automatically log out after a period of inactivity (see below), or you can make it happen with:

```
def logout(self, user):
```

There are three user states that are important to the manager:

- modified
- cached
- authenticated or "active"

A modified user is one whose data has changed and eventually requires storage to a persistent location. A cached user is a user whose data resides in memory (regardless of the other states). An active user has been authenticated (e.g., their username and password were checked) and has not yet logged out or timed out.

The manager keeps three timeouts, expressed in minutes, to:

- save modified users after a period of time following the first unsaved modification

- push users out of memory after a period of inactivity
- deactivate (e.g., log out) users after a period of inactivity

The methods for managing these values deal with the timeouts as number-of-minutes. The default values and the methods are:

- 20 modifiedUserTimeout() setModifiedUserTimeout()
- 20 cachedUserTimeout() setCachedUserTimeout()
- 20 activeUserTimeout() setActiveUserTimeout()

Subclasses of UserManager provide persistence such as to the file system or a MiddleKit store. Subclasses must implement all methods that raise AbstractErrors. Subclasses typically override (while still invoking super) addUser().

Subclasses should ensure "uniqueness" of users. For example, invoking any of the userForSomething() methods repeatedly should always return the same user instance for a given key. Without uniqueness, consistency issues could arise with users that are modified.

Please read the method docstrings and other class documentation to fully understand UserKit.

`__init__`(*userClass=None*)

`activeUserTimeout`()

`activeUsers`()

Return a list of all active users.

`addUser`(*user*)

`cachedUserTimeout`()

`clearCache`()

Clear the cache of the manager.

Use with extreme caution. If your program maintains a reference to a user object, but the manager loads in a new copy later on, then consistency problems could occur.

The most popular use of this method is in the regression test suite.

`createUser`(*name*, *password*, *userClass=None*)

Return a newly created user that is added to the manager.

If userClass is not specified, the manager's default user class is instantiated. This not imply that the user is logged in. This method invokes self.addUser().

See also: userClass(), setUserClass()

`inactiveUsers`()

`login`(*user*, *password*)

Return the user if login is successful, otherwise return None.

`loginExternalId`(*externalId*, *password*)

`loginName`(*userName*, *password*)

`loginSerialNum`(*serialNum*, *password*)

`logout`(*user*)

`modifiedUserTimeout``()`

`numActiveUsers``()`

Return the number of active users (e.g. the logged in users).

`setActiveUserTimeout``(`*value*`)`

`setCachedUserTimeout``(`*value*`)`

`setModifiedUserTimeout``(`*value*`)`

`setUserClass``(`*userClass*`)`

Set the userClass, which cannot be None and must inherit from User.

See also: userClass().

`shutDown``()`

Perform any tasks necessary to shut down the user manager.

Subclasses may override and must invoke super as their *last* step.

`userClass``()`

Return the userClass, which is used by createUser.

The default value is UserKit.User.User.

`userForExternalId``(`*externalId*, *default=<class 'MiscUtils.NoDefault'>*`)`

Return the user with the given external id.

The user record is pulled into memory if needed.

`userForName``(`*name*, *default=<class 'MiscUtils.NoDefault'>*`)`

Return the user with the given name.

The user record is pulled into memory if needed.

`userForSerialNum``(`*serialNum*, *default=<class 'MiscUtils.NoDefault'>*`)`

Return the user with the given serialNum.

The user record is pulled into memory if needed.

`users``()`

Return a list of all users (regardless of login status).

19.3.9 UserManagerToFile

The UserManagerToFile class.

class `UserKit.UserManagerToFile.`**`UserManagerToFile`**`(`*userClass=None*`)`

Bases: *`UserManager`*

User manager storing user data in the file system.

When using this user manager, make sure you invoke setUserDir() and that this directory is writeable by your application. It will contain one file per user with the user's serial number as the main filename and an extension of '.user'.

The default user directory is the current working directory, but relying on the current directory is often a bad practice.

`__init__`(*userClass=None*)

`activeUserTimeout`()

`activeUsers`()

Return a list of all active users.

`addUser`(*user*)

`cachedUserTimeout`()

`clearCache`()

Clear the cache of the manager.

Use with extreme caution. If your program maintains a reference to a user object, but the manager loads in a new copy later on, then consistency problems could occur.

The most popular use of this method is in the regression test suite.

`createUser`(*name*, *password*, *userClass=None*)

Return a newly created user that is added to the manager.

If userClass is not specified, the manager's default user class is instantiated. This not imply that the user is logged in. This method invokes self.addUser().

See also: userClass(), setUserClass()

`decoder`()

`encoder`()

`inactiveUsers`()

`initNextSerialNum`()

`loadUser`(*serialNum*, *default=<class 'MiscUtils.NoDefault'>*)

Load the user with the given serial number from disk.

If there is no such user, a KeyError will be raised unless a default value was passed, in which case that value is returned.

`login`(*user*, *password*)

Return the user if login is successful, otherwise return None.

`loginExternalId`(*externalId*, *password*)

`loginName`(*userName*, *password*)

`loginSerialNum`(*serialNum*, *password*)

`logout`(*user*)

`modifiedUserTimeout`()

`nextSerialNum`()

`numActiveUsers`()

Return the number of active users (e.g. the logged in users).

scanSerialNums()

Return a list of all the serial numbers of users found on disk.

Serial numbers are always integers.

setActiveUserTimeout(*value*)

setCachedUserTimeout(*value*)

setEncoderDecoder(*encoder*, *decoder*)

setModifiedUserTimeout(*value*)

setUserClass(*userClass*)

Overridden to mix in UserMixIn to the class that is passed in.

setUserDir(*userDir*)

Set the directory where user information is stored.

You should strongly consider invoking initNextSerialNum() afterwards.

shutDown()

Perform any tasks necessary to shut down the user manager.

Subclasses may override and must invoke super as their *last* step.

userClass()

Return the userClass, which is used by createUser.

The default value is UserKit.User.User.

userDir()

userForExternalId(*externalId*, *default=<class 'MiscUtils.NoDefault'>*)

Return the user with the given external id.

The user record is pulled into memory if needed.

userForName(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return the user with the given name.

The user record is pulled into memory if needed.

userForSerialNum(*serialNum*, *default=<class 'MiscUtils.NoDefault'>*)

Return the user with the given serialNum.

The user record is pulled into memory if needed.

users()

Return a list of all users (regardless of login status).

class UserKit.UserManagerToFile.**UserMixIn**

Bases: `object`

filename()

save()

19.4 TaskKit

19.4.1 Scheduler

This is the task manager Python package.

It provides a system for running any number of predefined tasks in separate threads in an organized and controlled manner.

A task in this package is a class derived from the Task class. The task should have a run method that, when called, performs some task.

The Scheduler class is the organizing object. It manages the addition, execution, deletion, and well being of a number of tasks. Once you have created your task class, you call the Scheduler to get it added to the tasks to be run.

class `TaskKit.Scheduler.`**`Scheduler`**(*daemon=True*, *exceptionHandler=None*)

Bases: `Thread`

The top level class of the task manager system.

The Scheduler is a thread that handles organizing and running tasks. The Scheduler class should be instantiated to start a task manager session. Its start method should be called to start the task manager. Its stop method should be called to end the task manager session.

`__init__`(*daemon=True*, *exceptionHandler=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

`addActionOnDemand`(*task*, *name*)

Add a task to be run only on demand.

Adds a task to the scheduler that will not be scheduled until specifically requested.

`addDailyAction`(*hour*, *minute*, *task*, *name*)

Add an action to be run every day at a specific time.

If a task with the given name is already registered with the scheduler, that task will be removed from the scheduling queue and registered anew as a periodic task.

Can we make this addCalendarAction? What if we want to run something once a week? We probably don't need that for Webware, but this is a more generally useful module. This could be a difficult function, though. Particularly without mxDateTime.

`addPeriodicAction`(*start*, *period*, *task*, *name*)

Add a task to be run periodically.

Adds an action to be run at a specific initial time, and every period thereafter.

The scheduler will not reschedule a task until the last scheduled instance of the task has completed.

If a task with the given name is already registered with the scheduler, that task will be removed from the scheduling queue and registered anew as a periodic task.

`addTimedAction`(*actionTime*, *task*, *name*)

Add a task to be run once, at a specific time.

`property daemon`

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

`delOnDemand`(*name*)

Delete a task with the given name from the on demand list.

`delRunning`(*name*)

Delete a task from the running list.

Used internally.

`delScheduled`(*name*)

Delete a task with the given name from the scheduled list.

`demandTask`(*name*)

Demand execution of a task.

Allow the server to request that a task listed as being registered on-demand be run as soon as possible.

If the task is currently running, it will be flagged to run again as soon as the current run completes.

Returns False if the task name could not be found on the on-demand or currently running lists.

`disableTask`(*name*)

Specify that a task be suspended.

Suspended tasks will not be scheduled until later enabled. If the task is currently running, it will not be interfered with, but the task will not be scheduled for execution in future until re-enabled.

Returns True if the task was found and disabled.

`enableTask`(*name*)

Enable a task again.

This method is provided to specify that a task be re-enabled after a suspension. A re-enabled task will be scheduled for execution according to its original schedule, with any runtimes that would have been issued during the time the task was suspended simply skipped.

Returns True if the task was found and enabled.

`getName`()

`hasOnDemand`(*name*)

Checks whether task with given name is in the on demand list.

`hasRunning`(*name*)

Check to see if a task with the given name is currently running.

`hasScheduled`(*name*)

Checks whether task with given name is in the scheduled list.

`property ident`

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

`isAlive()`

Return whether the thread is alive.

This method is deprecated, use is_alive() instead.

`isDaemon()`

`isRunning()`

Check whether thread is running.

`is_alive()`

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

`join(`*timeout=None*`)`

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

`property name`

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

`nextTime()`

Get next execution time.

`notify()`

Wake up scheduler by sending a notify even.

`notifyCompletion(`*handle*`)`

Notify completion of a task.

Used by instances of TaskHandler to let the Scheduler thread know when their tasks have run to completion. This method is responsible for rescheduling the task if it is a periodic task.

`notifyFailure(`*handle*`)`

Notify failure of a task.

Used by instances of TaskHandler to let the Scheduler thread know if an exception has occurred within the task thread.

`onDemand`(*name*, *default=None*)

Return a task from the onDemand list.

`onDemandTasks`()

Return all on demand tasks.

`run`()

The main method of the scheduler running as a background thread.

This method is responsible for carrying out the scheduling work of this class on a background thread. The basic logic is to wait until the next action is due to run, move the task from our scheduled list to our running list, and run it. Other synchronized methods such as runTask(), scheduleTask(), and notifyCompletion(), may be called while this method is waiting for something to happen. These methods modify the data structures that run() uses to determine its scheduling needs.

`runTask`(*handle*)

Run a task.

Used by the Scheduler thread's main loop to put a task in the scheduled hash onto the run hash.

`runTaskNow`(*name*)

Allow a registered task to be immediately executed.

Returns True if the task is either currently running or was started, or False if the task could not be found in the list of currently registered tasks.

`running`(*name*, *default=None*)

Return running task with given name.

Returns a task with the given name from the "running" list, if it is present there.

`runningTasks`()

Return all running tasks.

`scheduleTask`(*handle*)

Schedule a task.

This method takes a task that needs to be scheduled and adds it to the scheduler. All scheduling additions or changes are handled by this method. This is the only Scheduler method that can notify the run() method that it may need to wake up early to handle a newly registered task.

`scheduled`(*name*, *default=None*)

Return a task from the scheduled list.

`scheduledTasks`()

Return all scheduled tasks.

`setDaemon`(*daemonic*)

`setName`(*name*)

`setNextTime`(*nextTime*)

Set next execution time.

`setOnDemand`(*handle*)

Add the given task to the on demand list.

setRunning(*handle*)
Add a task to the running dictionary.

Used internally only.

setScheduled(*handle*)
Add the given task to the scheduled list.

start()
Start the scheduler's activity.

stop()
Terminate the scheduler and its associated tasks.

stopAllTasks()
Terminate all running tasks.

stopTask(*name*)
Put an immediate halt to a running background task.

Returns True if the task was either not running, or was running and was told to stop.

unregisterTask(*name*)
Unregisters the named task.

After that it can be rescheduled with different parameters, or simply removed.

wait(*seconds=None*)
Our own version of wait().

When called, it waits for the specified number of seconds, or until it is notified that it needs to wake up, through the notify event.

19.4.2 Task

Base class for tasks.

class TaskKit.Task.**Task**
Bases: `object`

Abstract base class from which you have to derive your own tasks.

__init__()
Subclasses should invoke super for this method.

handle()
Return the task handle.

A task is scheduled by wrapping a handler around it. It knows everything about the scheduling (periodicity and the like). Under normal circumstances you should not need the handler, but if you want to write period modifying run() methods, it is useful to have access to the handler. Use it with care.

name()
Return the unique name under which the task was scheduled.

proceed()
Check whether this task should continue running.

Should be called periodically by long tasks to check if the system wants them to exit. Returns True if its OK to continue, False if it's time to quit.

run()

Override this method for you own tasks.

Long running tasks can periodically use the proceed() method to check if a task should stop.

19.4.3 TaskHandler

class TaskKit.TaskHandler.**TaskHandler**(*scheduler*, *start*, *period*, *task*, *name*)

Bases: `object`

The task handler.

While the Task class only knows what task to perform with the run()-method, the TaskHandler has all the knowledge about the periodicity of the task. Instances of this class are managed by the Scheduler in the scheduled, running and onDemand dictionaries.

__init__(*scheduler*, *start*, *period*, *task*, *name*)

disable()

Disable future invocations of this task.

enable()

Enable future invocations of this task.

isOnDemand()

Return True if this task is not scheduled for periodic execution.

isRunning()

name()

notifyCompletion()

notifyFailure()

period()

Return the period of this task.

reschedule()

Determine whether this task should be rescheduled.

Increments the startTime and returns true if this is a periodically executed task.

reset(*start*, *period*, *task*, *reRegister*)

runAgain()

Determine whether this task should be run again.

This method lets the Scheduler check to see whether this task should be re-run when it terminates.

runOnCompletion()

Request that this task be re-run after its current completion.

Intended for on-demand tasks that are requested by the Scheduler while they are already running.

runTask()

Run this task in a background thread.

setOnDemand(*onDemand=True*)

setPeriod(*period*)

Change the period for this task.

startTime(*newTime=None*)

stop()

unregister()

Request that this task not be kept after its current completion.

Used to remove a task from the scheduler.

19.5 WebUtils

19.5.1 ExpansiveHTMLForException

ExpansiveHTMLForException.py

Create expansive HTML for exceptions using the CGITraceback module.

`WebUtils.ExpansiveHTMLForException.`**ExpansiveHTMLForException**(*context=5*, *options=None*)

Create expansive HTML for exceptions.

`WebUtils.ExpansiveHTMLForException.`**expansiveHTMLForException**(*context=5*, *options=None*)

Create expansive HTML for exceptions.

19.5.2 FieldStorage

FieldStorage.py

This module provides that latest version of the now deprecated standard Python cgi.FieldStorage class with a slight modification so that fields passed in the body of a POST request override any fields passed in the query string.

class `WebUtils.FieldStorage.`**FieldStorage**(*fp=None*, *headers=None*, *outerboundary=b''*, *environ=None*, *keep_blank_values=False*, *strict_parsing=False*, *limit=None*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Bases: `object`

Store a sequence of fields, reading multipart/form-data.

This is a slightly modified version of the FieldStorage class in the now deprecated cgi module of the standard library.

This class provides naming, typing, files stored on disk, and more. At the top level, it is accessible like a dictionary, whose keys are the field names. (Note: None can occur as a field name.) The items are either a Python list (if there's multiple values) or another FieldStorage or MiniFieldStorage object. If it's a single object, it has the following attributes:

name: the field name, if specified; otherwise None filename: the filename, if specified; otherwise None; this is the client side filename, *not* the file name on which it is stored (that's a temporary file you don't deal with)

value: the value as a *string*; for file uploads, this transparently reads the file every time you request the value and returns *bytes*

file: the file(-like) object from which you can read the data *as bytes*; None if the data is stored a simple string type: the content-type, or None if not specified

type_options: dictionary of options specified on the content-type line

disposition: content-disposition, or None if not specified

disposition_options: dictionary of corresponding options

headers: a dictionary(-like) object (sometimes email.message.Message or a subclass thereof) containing *all* headers

The class can be subclassed, mostly for the purpose of overriding the make_file() method, which is called internally to come up with a file open for reading and writing. This makes it possible to override the default choice of storing all files in a temporary directory and unlinking them as soon as they have been opened.

Parameters in the query string which have not been sent via POST are appended to the field list. This is different from the behavior of Python versions before 2.6 which completely ignored the query string in POST request, but it's also different from the behavior of the later Python versions which append values from the query string to values sent via POST for parameters with the same name. With other words, our FieldStorage class overrides the query string parameters with the parameters sent via POST.

`FieldStorageClass = None`

`__init__`(*fp=None*, *headers=None*, *outerboundary=b''*, *environ=None*, *keep_blank_values=False*, *strict_parsing=False*, *limit=None*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Constructor. Read multipart/* until last part. Arguments, all optional: fp: file pointer; default: sys.stdin.buffer

Not used when the request method is GET. Can be a TextIOWrapper object or an object whose read() and readline() methods return bytes.

headers: header dictionary-like object; default: taken from environ as per CGI spec

outerboundary: terminating multipart boundary (for internal use only)

environ: environment dictionary; default: os.environ

keep_blank_values: flag indicating whether blank values in percent-encoded forms should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

strict_parsing: flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a ValueError exception.

limit: used internally to read parts of multipart/form-data forms, to exit from the reading loop when reached. It is the difference between the form content-length and the number of bytes already read.

encoding, errors: the encoding and error handler used to decode the binary stream to strings. Must be the same as the charset defined for the page sending the form (content-type : meta http-equiv or header)

max_num_fields: int. If set, then __init__ throws a ValueError if there are more than n fields read by parse_qsl().

`bufsize = 8192`

`getfirst`(*key*, *default=None*)

Return the first value received.

`getlist`(*key*)

Return list of received values.

`getvalue`(*key*, *default=None*)

Dictionary style get() method, including ‘value’ lookup.

`keys`()

Dictionary style keys() method.

`make_file`()

Overridable: return a readable & writable file.

The file will be used as follows: - data is written to it - seek(0) - data is read from it The file is opened in binary mode for files, in text mode for other fields. This version opens a temporary file for reading and writing, and immediately deletes (unlinks) it. The trick (on Unix!) is that the file can still be used, but it can’t be opened by another process, and it will automatically be deleted when it is closed or when the current process terminates. If you want a more permanent file, you derive a class which overrides this method. If you want a visible temporary file that is nevertheless automatically deleted when the script terminates, try defining a __del__ method in a derived class which unlinks the temporary files you have created.

`read_binary`()

Internal: read binary data.

`read_lines`()

Internal: read lines until EOF or outerboundary.

`read_lines_to_eof`()

Internal: read lines until EOF.

`read_lines_to_outerboundary`()

Internal: read lines until outerboundary.

Data is read as bytes: boundaries and line ends must be converted to bytes for comparisons.

`read_multi`(*environ*, *keep_blank_values*, *strict_parsing*)

Internal: read a part that is itself multipart.

`read_single`()

Internal: read an atomic part.

`read_urlencoded`()

Internal: read data in query string format.

`skip_lines`()

Internal: skip lines until outer boundary if defined.

class `WebUtils.FieldStorage.MiniFieldStorage`(*name*, *value*)

Bases: `object`

Like FieldStorage, for use when no file uploads are possible.

`__init__`(*name*, *value*)

Constructor from field name and value.

`disposition = None`

`disposition_options = {}`

`file = None`

`filename = None`

headers = {}

list = None

type = None

type_options = {}

WebUtils.FieldStorage.**hasSeparator**()

Check whether the separator parameter is supported.

WebUtils.FieldStorage.**isBinaryType**(*ctype*, *pdict=None*)

“Check whether the given MIME type uses binary data.

WebUtils.FieldStorage.**parse_header**(*line*)

Parse a Content-type like header.

Return the main content-type and a dictionary of options.

WebUtils.FieldStorage.**valid_boundary**(*s*)

19.5.3 Funcs

WebUtils.Funcs

This module provides some basic functions that are useful in HTML and web development.

You can safely import * from WebUtils.Funcs if you like.

WebUtils.Funcs.**htmlDecode**(*s*, *codes=(('"', '&quot;'), ('>', '&gt;'), ('<', '&lt;'), ('&', '&amp;'))*)

Return the ASCII decoded version of the given HTML string.

This does NOT remove normal HTML tags like <p>. It is the inverse of htmlEncode().

The optional ‘codes’ parameter allows passing custom translations.

WebUtils.Funcs.**htmlEncode**(*what*, *codes=(('&', '&amp;'), ('<', '&lt;'), ('>', '&gt;'), ('"', '&quot;'))*)

Return the HTML encoded version of the given object.

The optional ‘codes’ parameter allows passing custom translations.

WebUtils.Funcs.**htmlEncodeStr**(*s*, *codes=(('&', '&amp;'), ('<', '&lt;'), ('>', '&gt;'), ('"', '&quot;'))*)

Return the HTML encoded version of the given string.

This is useful to display a plain ASCII text string on a web page.

The optional ‘codes’ parameter allows passing custom translations.

WebUtils.Funcs.**htmlForDict**(*d*, *addSpace=None*, *filterValueCallBack=None*, *maxValueLength=None*, *topHeading=None*, *isEncoded=None*)

Return HTML string with a table where each row is a key/value pair.

WebUtils.Funcs.**normURL**(*path*)

Normalizes a URL path, like os.path.normpath.

Acts on a URL independent of operating system environment.

`WebUtils.Funcs.`**`requestURI`**`(`*env*`)`

Return the request URI for a given CGI-style dictionary.

Uses REQUEST_URI if available, otherwise constructs and returns it from SCRIPT_URL, SCRIPT_NAME, PATH_INFO and QUERY_STRING.

`WebUtils.Funcs.`**`urlDecode`**`(`*string*, *encoding='utf-8'*, *errors='replace'*`)`

Like unquote(), but also replace plus signs by spaces, as required for unquoting HTML form values.

unquote_plus('%7e/abc+def') -> '~/abc def'

`WebUtils.Funcs.`**`urlEncode`**`(`*string*, *safe=''*, *encoding=None*, *errors=None*`)`

Like quote(), but also replace ' ' with '+', as required for quoting HTML form values. Plus signs in the original string are escaped unless they are included in safe. It also does not have safe default to '/'.

19.5.4 HTMLForException

HTMLForException.py

Create HTML for exceptions.

`WebUtils.HTMLForException.`**`HTMLForException`**`(`*excInfo=None*, *options=None*`)`

Get HTML for displaying an exception.

Returns an HTML string that presents useful information to the developer about the exception. The first argument is a tuple such as returned by sys.exc_info() which is in fact invoked if the tuple isn't provided.

`WebUtils.HTMLForException.`**`HTMLForLines`**`(`*lines*, *options=None*`)`

Create HTML for exceptions and tracebacks from a list of strings.

`WebUtils.HTMLForException.`**`HTMLForStackTrace`**`(`*frame=None*, *options=None*`)`

Get HTML for displaying a stack trace.

Returns an HTML string that presents useful information to the developer about the stack. The first argument is a stack frame such as returned by sys._getframe() which is in fact invoked if a stack frame isn't provided.

`WebUtils.HTMLForException.`**`htmlForException`**`(`*excInfo=None*, *options=None*`)`

Get HTML for displaying an exception.

Returns an HTML string that presents useful information to the developer about the exception. The first argument is a tuple such as returned by sys.exc_info() which is in fact invoked if the tuple isn't provided.

`WebUtils.HTMLForException.`**`htmlForLines`**`(`*lines*, *options=None*`)`

Create HTML for exceptions and tracebacks from a list of strings.

`WebUtils.HTMLForException.`**`htmlForStackTrace`**`(`*frame=None*, *options=None*`)`

Get HTML for displaying a stack trace.

Returns an HTML string that presents useful information to the developer about the stack. The first argument is a stack frame such as returned by sys._getframe() which is in fact invoked if a stack frame isn't provided.

19.5.5 HTMLTag

HTMLTag.py

HTMLTag defines a class of the same name that represents HTML content. An additional HTMLReader class kicks off the process of reading an HTML file into a set of tags:

```
from WebUtils.HTMLTag import HTMLReader
reader = HTMLReader()
tag = reader.readFileNamed('foo.html')
tag.pprint()
```

Tags have attributes and children, which makes them hierarchical. See HTMLTag class docs for more info.

Note that you imported HTMLReader instead of HTMLTag. You only need the latter if you plan on creating tags directly.

You can discard the reader immediately if you like:

```
tag = HTMLReader().readFileNamed('foo.html')
```

The point of reading HTML into tag objects is so that you have a concrete, Pythonic data structure to work with. The original motivation for such a beast was in building automated regression test suites that wanted granular, structured access to the HTML output by the web application.

See the doc string for HTMLTag for examples of what you can do with tags.

exception `WebUtils.HTMLTag.`**`HTMLNotAllowedError`**(*msg*, ***values*)

Bases: *`HTMLTagError`*

HTML tag not allowed here error

`__init__`(*msg*, ***values*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `WebUtils.HTMLTag.`**`HTMLReader`**(*emptyTags=None*, *extraEmptyTags=None*, *fakeRootTagIfNeeded=True*)

Bases: `HTMLParser`

Reader class for representing HTML as tag objects.

NOTES

- Special attention is required regarding tags like `<p>` and `<li>` which sometimes are closed and sometimes not. HTMLReader can deal with both situations (closed and not) provided that:
  - the file doesn't change conventions for a given tag
  - the reader knows ahead of time what to expect

Be default, *HTMLReader* assumes that `<p>` and `<li>` will be closed with `</p>` and `</li>` as the official HTML spec encourages.

But if your files don't close certain tags that are supposed to be required, you can do this:

```
HTMLReader(extraEmptyTags=['p', 'li'])
```

or:

```
reader.extendEmptyTags(['p', 'li'])
```

or just set them entirely:

```
HTMLReader(emptyTags=['br', 'hr', 'p'])
reader.setEmptyTags(['br', 'hr', 'p'])
```

Although there are quite a few. Consider the DefaultEmptyTags global list (which is used to initialize the reader's tags) which contains about 16 tag names.

If an HTML file doesn't conform to the reader's expectation, you will get an exception (see more below for details).

If your HTML file doesn't contain root `<html> ... </html>` tags wrapping everything, a fake root tag will be constructed for you, unless you pass in fakeRootTagIfNeeded=False.

Besides fixing your reader manually, you could conceivably loop through the permutations of the various empty tags to see if one of them resulted in a correct read.

Or you could fix the HTML.

- The reader ignores extra preceding and trailing whitespace by stripping it from strings. I suppose this is a little harsher than reducing spans of preceding and trailing whitespace down to one space, which is what really happens in an HTML browser.
- The reader will not read past the closing `</html>` tag.
- The reader is picky about the correctness of the HTML you feed it. If tags are not closed, overlap (instead of nest) or left unfinished, an exception is thrown. These include *HTMLTagUnbalancedError*, *HTMLTagIncompleteError* and *HTMLNotAllowedError* which all inherit *HTMLTagError*.

  This pickiness can be quite useful for the validation of the HTML of your own applications.

`CDATA_CONTENT_ELEMENTS = ('script', 'style')`

`__init__`(*emptyTags=None*, *extraEmptyTags=None*, *fakeRootTagIfNeeded=True*)

Initialize and reset this instance.

If convert_charrefs is True (the default), all character references are automatically converted to the corresponding Unicode characters.

`check_for_whole_start_tag`(*i*)

`clear_cdata_mode`()

`close`()

Handle any buffered data.

`computeTagContainmentConfig`()

`emptyTags`()

Return a list of empty tags.

See also: class docs and setEmptyTags().

`error`(*message*)

`extendEmptyTags`(*tagList*)

Extend the current list of empty tags with the given list.

`feed`(*data*)

Feed data to the parser.

Call this as often as you want, with as little or as much text as you want (may include 'n').

`filename`()

Return the name of the file if one has been read, otherwise None.

`get_starttag_text`()

Return full source of start tag: '<. . . >'.

`getpos`()

Return current line number and offset.

`goahead`(*end*)

`handle_charref`(*name*)

`handle_comment`(*data*)

`handle_data`(*data*)

`handle_decl`(*decl*)

`handle_endtag`(*tag*)

`handle_entityref`(*name*)

`handle_pi`(*data*)

`handle_startendtag`(*tag*, *attrs*)

`handle_starttag`(*tag*, *attrs*)

`main`(*args=None*)

The command line equivalent of readFileNamed().

Invoked when HTMLTag is run as a program.

`parse_bogus_comment`(*i*, *report=1*)

`parse_comment`(*i*, *report=1*)

`parse_declaration`(*i*)

`parse_endtag`(*i*)

`parse_html_declaration`(*i*)

`parse_marked_section`(*i*, *report=1*)

`parse_pi`(*i*)

`parse_starttag`(*i*)

`pprint`(*out=None*)

Pretty prints the tag, its attributes and all its children.

Indentation is used for subtags. Print 'Empty.' if there is no root tag.

`printsStack`()

readFileNamed(*filename*, *retainRootTag=True*, *encoding='utf-8'*)

Read the given file.

Relies on readString(). See that method for more information.

readString(*string*, *retainRootTag=True*)

Read the given string, store the results and return the root tag.

You could continue to use HTMLReader object or disregard it and simply use the root tag.

reset()

Reset this instance. Loses all unprocessed data.

rootTag()

Return the root tag.

May return None if no HTML has been read yet, or if the last invocation of one of the read methods was passed retainRootTag=False.

setEmptyTags(*tagList*)

Set the HTML tags that are considered empty such as <br> and <hr>.

The default is found in the global, DefaultEmptyTags, and is fairly thorough, but does not include <p>, <li> and some other tags that HTML authors often use as empty tags.

setPrintsStack(*flag*)

Set the boolean value of the "prints stack" option.

This is a debugging option which will print the internal tag stack during HTML processing. The default value is False.

set_cdata_mode(*elem*)

tagContainmentConfig = {'body': 'cannotHave html head body', 'head': 'cannotHave html head body', 'html': 'canOnlyHave head body', 'select': 'canOnlyHave option', 'table': 'canOnlyHave tr thead tbody tfoot a', 'td': 'cannotHave td tr', 'tr': 'canOnlyHave th td'}

unescape(*s*)

unknown_decl(*data*)

updatepos(*i*, *j*)

usage()

class WebUtils.HTMLTag.**HTMLTag**(*name*, *lineNumber=None*)

Bases: `object`

Container class for representing HTML as tag objects.

Tags essentially have 4 major attributes:

- name
- attributes
- children
- subtags

Name is simple:
print(tag.name())

Attributes are dictionary-like in nature:

```
print(tag.attr('color'))  # throws an exception if no color
print(tag.attr('bgcolor', None))  # returns None if no bgcolor
print(tag.attrs())
```

Children are all the leaf parts of a tag, consisting of other tags and strings of character data:

```
print(tag.numChildren())
print(tag.childAt(0))
print(tag.children())
```

Subtags is a convenient list of only the tags in the children:

```
print(tag.numSubtags())
print(tag.subtagAt(0))
print(tag.subtags())
```

You can search a tag and all the tags it contains for a tag with a particular attribute matching a particular value:

```
print(tag.tagWithMatchingAttr('width', '100%'))
```

An HTMLTagAttrLookupError is raised if no matching tag is found. You can avoid this by providing a default value:

```
print(tag.tagWithMatchingAttr('width', '100%', None))
```

Looking for specific 'id' attributes is common in regression testing (it allows you to zero in on logical portions of a page), so a convenience method is provided:

```
tag = htmlTag.tagWithId('accountTable')
```

`__init__`(*name*, *lineNumber=None*)

`addChild`(*child*)

Add a child to the receiver.

The child will be another tag or a string (CDATA).

`attr`(*name*, *default=<class 'MiscUtils.NoDefault'>*)

`attrs`()

`childAt`(*index*)

`children`()

`closedBy`(*name*, *lineNumber*)

`hasAttr`(*name*)

`name`()

`numAttrs`()

`numChildren`()

`numSubtags`()

pprint(*out=None*, *indent=0*)

readAttr(*name*, *value*)

Set an attribute of the tag with the given name and value.

A HTMLTagAttrLookupError is raised if an attribute is set twice.

subtagAt(*index*)

subtags()

tagWithId(*id_*, *default=<class 'MiscUtils.NoDefault'>*)

Search for tag with a given id.

Finds and returns the tag with the given id. As in:

```
<td id=foo> bar </td>
```

This is just a cover for:

```
tagWithMatchingAttr('id', id_, default)
```

But searching for id's is so popular (at least in regression testing web sites) that this convenience method is provided. Why is it so popular? Because by attaching ids to logical portions of your HTML, your regression test suite can quickly zero in on them for examination.

tagWithMatchingAttr(*name*, *value*, *default=<class 'MiscUtils.NoDefault'>*)

Search for tag with matching attributes.

Performs a depth-first search for a tag with an attribute that matches the given value. If the tag cannot be found, a KeyError will be raised *unless* a default value was specified, which is then returned.

Example:

```
tag = tag.tagWithMatchingAttr('bgcolor', '#FFFFFF', None)
```

exception WebUtils.HTMLTag.**HTMLTagAttrLookupError**(*msg*, ***values*)

Bases: *HTMLTagError*, LookupError

HTML tag attribute lookup error

__init__(*msg*, ***values*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception WebUtils.HTMLTag.**HTMLTagError**(*msg*, ***values*)

Bases: Exception

General HTML tag error

__init__(*msg*, ***values*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `WebUtils.HTMLTag.`**`HTMLTagIncompleteError`**(*msg*, ***values*)

Bases: *`HTMLTagError`*

HTML tag incomplete error

`__init__`(*msg*, ***values*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `WebUtils.HTMLTag.`**`HTMLTagProcessingInstructionError`**(*msg*, ***values*)

Bases: *`HTMLTagError`*

HTML tag processing instruction error

`__init__`(*msg*, ***values*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `WebUtils.HTMLTag.`**`HTMLTagUnbalancedError`**(*msg*, ***values*)

Bases: *`HTMLTagError`*

Unbalanced HTML tag error

`__init__`(*msg*, ***values*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `WebUtils.HTMLTag.`**`TagCanOnlyHaveConfig`**(*name*, *tags*)

Bases: *`TagConfig`*

`__init__`(*name*, *tags*)

`encounteredTag`(*tag*, *lineNum*)

class `WebUtils.HTMLTag.`**`TagCannotHaveConfig`**(*name*, *tags*)

Bases: *`TagConfig`*

`__init__`(*name*, *tags*)

`encounteredTag`(*tag*, *lineNum*)

class `WebUtils.HTMLTag.`**`TagConfig`**(*name*, *tags*)

Bases: `object`

`__init__`(*name*, *tags*)

`encounteredTag`(*tag*, *lineNum*)

19.5.6 HTTPStatusCodes

HTTPStatusCodes.py

Dictionary of HTTP status codes.

`WebUtils.HTTPStatusCodes.`**`htmlTableOfHTTPStatusCodes`**`(`*codes=None*, *tableArgs=''*, *rowArgs='style="vertical-align:top"'*, *colArgs=''*, *headingArgs=''*`)`

Return an HTML table with HTTP status codes.

Returns an HTML string containing all the status code information as provided by this module. It's highly recommended that if you pass arguments to this function, that you do so by keyword.

19.6 MiscUtils

19.6.1 Configurable

Configurable.py

Provides configuration file functionality.

`class` `MiscUtils.Configurable.`**`Configurable`**

Bases: `object`

Abstract superclass for configuration file functionality.

Subclasses should override:

- **defaultConfig() to return a dictionary of default settings**
  such as {'Frequency': 5}
- **configFilename() to return the filename by which users can**
  override the configuration such as 'Pinger.config'

Subclasses typically use the setting() method, for example:

time.sleep(self.setting('Frequency'))

They might also use the printConfig() method, for example:

self.printConfig() # or self.printConfig(file)

Users of your software can create a file with the same name as configFilename() and selectively override settings. The format of the file is a Python dictionary.

Subclasses can also override userConfig() in order to obtain the user configuration settings from another source.

`__init__``()`

`commandLineConfig``()`

Return the settings that came from the command-line.

These settings come via addCommandLineSetting().

`config``()`

Return the configuration of the object as a dictionary.

This is a combination of defaultConfig() and userConfig(). This method caches the config.

configFilename()

Return the full name of the user config file.

Users can override the configuration by this config file. Subclasses must override to specify a name. Returning None is valid, in which case no user config file will be loaded.

configName()

Return the name of the configuration file without the extension.

This is the portion of the config file name before the '.config'. This is used on the command-line.

configReplacementValues()

Return a dictionary for substitutions in the config file.

This must be a dictionary suitable for use with "string % dict" that should be used on the text in the config file. If an empty dictionary (or None) is returned, then no substitution will be attempted.

defaultConfig()

Return a dictionary with all the default values for the settings.

This implementation returns {}. Subclasses should override.

hasSetting(*name*)

Check whether a configuration setting has been changed.

printConfig(*dest=None*)

Print the configuration to the given destination.

The default destination is stdout. A fixed with font is assumed for aligning the values to start at the same column.

static **readConfig**(*filename*)

Read the configuration from the file with the given name.

Raises an UIError if the configuration cannot be read.

This implementation assumes the file is stored in utf-8 encoding with possible BOM at the start, but also tries to read as latin-1 if it cannot be decoded as utf-8. Subclasses can override this behavior.

setSetting(*name*, *value*)

Set a particular configuration setting.

setting(*name*, *default=<class 'MiscUtils.NoDefault'>*)

Return the value of a particular setting in the configuration.

userConfig()

Return the user config overrides.

These settings can be found in the optional config file. Returns {} if there is no such file.

The config filename is taken from configFilename().

exception MiscUtils.Configurable.**ConfigurationError**

Bases: `Exception`

Error in configuration file.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

MiscUtils.Configurable.**addCommandLineSetting**(*name*, *value*)

Override the configuration with a command-line setting.

Take a setting, like "Application.Verbose=0", and call addCommandLineSetting('Application.Verbose', '0'), and it will override any settings in Application.config

MiscUtils.Configurable.**commandLineSetting**(*configName*, *settingName*, *default=<class 'MiscUtils.NoDefault'>*)

Retrieve a command-line setting.

You can use this with non-existent classes, like "Context.Root=/Webware", and then fetch it back with commandLineSetting('Context', 'Root').

19.6.2 CSVJoiner

CSVJoiner.py

A helper function for joining CSV fields.

MiscUtils.CSVJoiner.**joinCSVFields**(*fields*)

Create a CSV record by joining its fields.

Returns a CSV record (e.g. a string) from a sequence of fields. Fields containing commands (,) or double quotes (") are quoted, and double quotes are escaped (""). The terminating newline is *not* included.

19.6.3 CSVParser

CSVParser.py

A parser for CSV files.

class MiscUtils.CSVParser.**CSVParser**(*allowComments=True*, *stripWhitespace=True*, *fieldSep=','*, *autoReset=True*, *doubleQuote=True*)

Bases: object

Parser for CSV files.

Parses CSV files including all subtleties such as:

- commas in fields
- double quotes in fields
- embedded newlines in fields

Examples of programs that produce such beasts include MySQL and Excel.

For a higher-level, friendlier CSV class with many conveniences, see *DataTable* (which uses this class for its parsing).

Example:

```
records = []
parse = CSVParser().parse
for line in lines:
    results = parse(line)
```

(continues on next page)

(continued from previous page)

```
    if results is not None:
        records.append(results)
```

CREDIT

The algorithm was taken directly from the open source Python C-extension, csv: https://www.object-craft.com.au/projects/csv/

It would be nice to use the csv module when present, since it is substantially faster. Before that can be done, it needs to support *allowComments* and *stripWhitespace*, and pass the TestCSVParser.py test suite.

`__init__`(*allowComments=True*, *stripWhitespace=True*, *fieldSep=','*, *autoReset=True*, *doubleQuote=True*)

Create a new CSV parser.

***allowComments*:**
If true (the default), then comment lines using the Python comment marker are allowed.

***stripWhitespace*:**
If true (the default), then left and right whitespace is stripped off from all fields.

***fieldSep*:**
Defines the field separator string (a comma by default).

***autoReset*:**
If true (the default), recover from errors automatically.

***doubleQuote*:**
If true (the default), assume quotes in fields are escaped by appearing doubled.

`endQuotedField`(*c*)

`inField`(*c*)

`inQuotedField`(*c*)

`parse`(*line*)

Parse a single line and return a list of string fields.

Returns None if the CSV record contains embedded newlines and the record is not yet complete.

`quoteInField`(*c*)

`quoteInQuotedField`(*c*)

`reset`()

Reset the parser.

Resets the parser to a fresh state in order to recover from exceptions. But if autoReset is true (the default), this is done automatically.

`saveField`()

`startField`(*c*)

`startRecord`(*c*)

exception `MiscUtils.CSVParser.`**`ParseError`**

Bases: `Exception`

CSV file parse error.

`__init__`(**args*, ***kwargs*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

`MiscUtils.CSVParser.`**`parse`**(*line*)

Parse a single line and return a list of string fields.

Returns None if the CSV record contains embedded newlines and the record is not yet complete.

19.6.4 DataTable

DataTable.py

INTRODUCTION

This class is useful for representing a table of data arranged by named columns, where each row in the table can be thought of as a record:

```
name   phoneNumber
------ -----------
Chuck  893-3498
Bill   893-0439
John   893-5901
```

This data often comes from delimited text files which typically have well-defined columns or fields with several rows each of which can be thought of as a record.

Using a DataTable can be as easy as using lists and dictionaries:

```
table = DataTable('users.csv')
for row in table:
    print(row['name'], row['phoneNumber'])
```

Or even:

```
table = DataTable('users.csv')
for row in table:
    print('{name} {phoneNumber}'.format(**row))
```

The above print statement relies on the fact that rows can be treated like dictionaries, using the column headings as keys.

You can also treat a row like an array:

```
table = DataTable('something.tabbed', delimiter='	')
for row in table:
    for item in row:
        print(item, end=' ')
    print()
```

COLUMNS

Column headings can have a type specification like so:

```
name, age:int, zip:int
```

Possible types include string, int, float and datetime.

String is assumed if no type is specified but you can set that assumption when you create the table:

```
table = DataTable(headings, defaultType='float')
```

Using types like int and float will cause DataTable to actually convert the string values (perhaps read from a file) to these types so that you can use them in natural operations. For example:

```
if row['age'] > 120:
    self.flagData(row, 'age looks high')
```

As you can see, each row can be accessed as a dictionary with keys according the column headings. Names are case sensitive.

ADDING ROWS

Like Python lists, data tables have an append() method. You can append TableRecords, or you pass a dictionary, list or object, in which case a TableRecord is created based on given values. See the method docs below for more details.

FILES

By default, the files that DataTable reads from are expected to be comma-separated value files.

Limited comments are supported: A comment is any line whose very first character is a #. This allows you to easily comment out lines in your data files without having to remove them.

Whitespace around field values is stripped.

You can control all this behavior through the arguments found in the initializer and the various readFoo() methods:

```
...delimiter=',', allowComments=True, stripWhite=True
```

For example:

```
table = DataTable('foo.tabbed', delimiter='	',
    allowComments=False, stripWhite=False)
```

You should access these parameters by their name since additional ones could appear in the future, thereby changing the order.

If you are creating these text files, we recommend the comma-separated-value format, or CSV. This format is better defined than the tab delimited format, and can easily be edited and manipulated by popular spreadsheets and databases.

MICROSOFT EXCEL

On Microsoft Windows systems with Excel and the PyWin32 package (https://github.com/mhammond/pywin32), DataTable will use Excel (via COM) to read ".xls" files:

```
from MiscUtils import DataTable
assert DataTable.canReadExcel()
table = DataTable.DataTable('foo.xls')
```

With consistency to its CSV processing, DataTable will ignore any row whose first cell is '#' and strip surrounding whitespace around strings.

TABLES FROM SCRATCH

Here's an example that constructs a table from scratch:

```
table = DataTable(['name', 'age:int'])
table.append(['John', 80])
table.append({'name': 'John', 'age': 80})
print(table)
```

QUERIES

A simple query mechanism is supported for equality of fields:

```
matches = table.recordsEqualTo({'uid': 5})
if matches:
    for match in matches:
        print(match)
else:
    print('No matches.')
```

COMMON USES

- Programs can keep configuration and other data in simple comma- separated text files and use DataTable to access them. For example, a web site could read its sidebar links from such a file, thereby allowing people who don't know Python (or even HTML) to edit these links without having to understand other implementation parts of the site.
- Servers can use DataTable to read and write log files.

FROM THE COMMAND LINE

The only purpose in invoking DataTable from the command line is to see if it will read a file:

```
> python DataTable.py foo.csv
```

The data table is printed to stdout.

CACHING

DataTable uses "pickle caching" so that it can read .csv files faster on subsequent loads. You can disable this across the board with:

```
from MiscUtils.DataTable import DataTable
DataTable._usePickleCache = False
```

Or per instance by passing "usePickleCache=False" to the constructor.

See the docstring of PickleCache.py for more information.

MORE DOCS

Some of the methods in this module have worthwhile doc strings to look at. See below.

TO DO

- Allow callback parameter or setting for parsing CSV records.
- Perhaps TableRecord should inherit list and dict and override methods as appropriate?
- _types and _blankValues aren't really packaged, advertised or documented for customization by the user of this module.
- **DataTable:**
  - Parameterize the TextColumn class.

    - Parameterize the TableRecord class.
    - More list-like methods such as insert()
    - writeFileNamed() is flawed: it doesn't write the table column type
    - Should it inherit from list?

- Add error checking that a column name is not a number (which could cause problems).
- Reading Excel sheets with xlrd, not only with win32com.

class MiscUtils.DataTable.**DataTable**(*filenameOrHeadings=None*, *delimiter=','*, *allowComments=True*, *stripWhite=True*, *encoding=None*, *defaultType=None*, *usePickleCache=None*)

Bases: `object`

Representation of a data table.

See the doc string for this module.

__init__(*filenameOrHeadings=None*, *delimiter=','*, *allowComments=True*, *stripWhite=True*, *encoding=None*, *defaultType=None*, *usePickleCache=None*)

append(*obj*)

Append an object to the table.

If obj is not a TableRecord, then one is created, passing the object to initialize the TableRecord. Therefore, obj can be a TableRecord, list, dictionary or object. See TableRecord for details.

static **canReadExcel**()

commit()

createNameToIndexMap()

Create speed-up index.

Invoked by self to create the nameToIndexMap after the table's headings have been read/initialized.

dictKeyedBy(*key*)

Return a dictionary containing the contents of the table.

The content is indexed by the particular key. This is useful for tables that have a column which represents a unique key (such as a name, serial number, etc.).

filename()

hasHeading(*name*)

heading(*index*)

headings()

nameToIndexMap()

Speed-up index.

Table rows keep a reference to this map in order to speed up index-by-names (as in row['name']).

numHeadings()

readExcel(*worksheet=1*, *row=1*, *column=1*)

readFile(*file*, *delimiter=','*, *allowComments=True*, *stripWhite=True*)

readFileNamed(*filename*, *delimiter=','*, *allowComments=True*, *stripWhite=True*, *encoding=None*, *worksheet=1*, *row=1*, *column=1*)

readLines(*lines*, *delimiter=','*, *allowComments=True*, *stripWhite=True*)

readString(*string*, *delimiter=','*, *allowComments=True*, *stripWhite=True*)

recordsEqualTo(*record*)

save()

setHeadings(*headings*)

Set table headings.

Headings can be a list of strings (like ['name', 'age:int']) or a list of TableColumns or None.

writeFile(*file*)

Write the table out as a file.

This doesn't write the column types (like int) back out.

It's notable that a blank numeric value gets read as zero and written out that way. Also, values None are written as blanks.

writeFileNamed(*filename*, *encoding='utf-8'*)

exception `MiscUtils.DataTable.`**DataTableError**

Bases: `Exception`

Data table error.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `MiscUtils.DataTable.`**TableColumn**(*spec*)

Bases: `object`

Representation of a table column.

A table column represents a column of the table including name and type. It does not contain the actual values of the column. These are stored individually in the rows.

__init__(*spec*)

Initialize the table column.

The spec parameter is a string such as 'name' or 'name:type'.

name()

setType(*type_*)

Set the type (by a string containing the name) of the heading.

Usually invoked by DataTable to set the default type for columns whose types were not specified.

type()

valueForRawValue(*value*)

Set correct type for raw value.

The rawValue is typically a string or value already of the appropriate type. TableRecord invokes this method to ensure that values (especially strings that come from files) are the correct types (e.g., ints are ints and floats are floats).

class `MiscUtils.DataTable.`**TableRecord**(*table*, *values=None*, *headings=None*)

Bases: `object`

Representation of a table record.

__init__(*table*, *values=None*, *headings=None*)

Initialize table record.

Dispatches control to one of the other init methods based on the type of values. Values can be one of three things:

1. A TableRecord
2. A list
3. A dictionary
4. Any object responding to hasValueForKey() and valueForKey().

asDict()

Return a dictionary whose key-values match the table record.

asList()

Return a sequence whose values are the same as the record's.

The order of the sequence is the one defined by the table.

get(*key*, *default=None*)

has_key(*key*)

initFromDict(*values*)

initFromObject(*obj*)

Initialize from object.

The object is expected to response to hasValueForKey(name) and valueForKey(name) for each of the headings in the table. It's alright if the object returns False for hasValueForKey(). In that case, a "blank" value is assumed (such as zero or an empty string). If hasValueForKey() returns True, then valueForKey() must return a value.

initFromSequence(*values*)

items()

iteritems()

iterkeys()

itervalues()

keys()

valueForAttr(*attr*, *default=<class 'MiscUtils.NoDefault'>*)

valueForKey(*key*, *default=<class 'MiscUtils.NoDefault'>*)

values()

MiscUtils.DataTable.**canReadExcel**()

MiscUtils.DataTable.**main**(*args=None*)

19.6.5 DateInterval

DateInterval.py

Convert interval strings (in the form of 1w2d, etc) to seconds, and back again. Is not exactly about months or years (leap years in particular).

Accepts (y)ear, (b)month, (w)eek, (d)ay, (h)our, (m)inute, (s)econd.

Exports only timeEncode and timeDecode functions.

MiscUtils.DateInterval.**timeDecode**(*s*)

Decode a number in the format 1h4d3m (1 hour, 3 days, 3 minutes).

Decode the format into a number of seconds.

MiscUtils.DateInterval.**timeEncode**(*seconds*)

Encode a number of seconds (representing a time interval).

Encode the number into a form like 2d1h3s.

19.6.6 DateParser

DateParser.py

Convert string representations of dates to Python datetime objects.

If installed, we will use the python-dateutil package to parse dates, otherwise we try to use the strptime function in the Python standard library with several frequently used formats.

MiscUtils.DateParser.**parseDate**(*s*)

Return a date object corresponding to the given string.

MiscUtils.DateParser.**parseDateTime**(*s*)

Return a datetime object corresponding to the given string.

MiscUtils.DateParser.**parseTime**(*s*)

Return a time object corresponding to the given string.

19.6.7 DBPool

DBPool.py

Implements a pool of cached connections to a database for any DB-API 2 compliant database module. This should result in a speedup for persistent applications like Webware. The pool of connections is threadsafe regardless of whether the used DB-API 2 general has a threadsafety of 1 or 2.

For more information on the DB-API 2 specification, see PEP 249.

The idea behind DBPool is that it's completely seamless, so once you have established your connection, use it just as you would any other DB-API 2 compliant module. For example:

```
import pgdb  # import used DB-API 2 module
from MiscUtils.DBPool import DBPool
dbpool = DBPool(pgdb, 5, host=..., database=..., user=..., ...)
db = dbpool.connection()
```

Now use "db" exactly as if it were a pgdb connection. It's really just a proxy class.

db.close() will return the connection to the pool, not actually close it. This is so your existing code works nicely.

DBPool is actually intended to be a demonstration of concept not to be used in a productive environment. It is really a very simple solution with several drawbacks. For instance, pooled database connections which have become invalid are not automatically recovered. For a more sophisticated solution, please have a look at the DBUtils package.

CREDIT

- Contributed by Dan Green.
- Thread safety bug found by Tom Schwaller.
- Fixes by Geoff Talvola (thread safety in *_threadsafe_get_connection()*).
- Clean up by Chuck Esterbrook.
- Fix unthreadsafe functions which were leaking, Jay Love.
- Eli Green's webware-discuss comments were lifted for additional docs.
- Coding and comment clean-up by Christoph Zwerschke.

`class MiscUtils.DBPool.`**`DBPool`**`(`*dbapi*, *maxconnections*, **args*, ***kwargs*`)`

Bases: `object`

`__init__``(`*dbapi*, *maxconnections*, **args*, ***kwargs*`)`

Set up the database connection pool.

***dbapi*:**
the DB-API 2 compliant module you want to use

***maxconnections*:**
the number of connections cached in the pool

***args*, *kwargs*:**
the parameters that shall be used to establish the database connections using `dbapi.connect()`

`exception MiscUtils.DBPool.`**`DBPoolError`**

Bases: `Exception`

General database pooling error.

`__init__``(`**args*, ***kwargs*`)`

`args`

`with_traceback``()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

`exception MiscUtils.DBPool.`**`NotSupportedError`**

Bases: *`DBPoolError`*

Missing support from database module error.

`__init__``(`**args*, ***kwargs*`)`

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `MiscUtils.DBPool.`**PooledConnection**(*pool*, *con*)

Bases: `object`

A wrapper for database connections to help with DBPool.

You don't normally deal with this class directly, but use DBPool to get new connections.

__init__(*pool*, *con*)

close()

Close the pooled connection.

19.6.8 DictForArgs

DictForArgs.py

See the doc string for the dictForArgs() function.

Also, there is a test suite in Tests/TestDictForArgs.py

`MiscUtils.DictForArgs.`**DictForArgs**(*s*)

Build dictionary from arguments.

Takes an input such as:

```
x=3
name="foo"
first='john' last='doe'
required border=3
```

And returns a dictionary representing the same. For keys that aren't given an explicit value (such as 'required' above), the value is '1'.

All values are interpreted as strings. If you want ints and floats, you'll have to convert them yourself.

This syntax is equivalent to what you find in HTML and close to other ML languages such as XML.

Returns { } for an empty string.

The informal grammar is:

```
(NAME [=NAME|STRING])*
```

Will raise DictForArgsError if the string is invalid.

See also: pyDictForArgs() and expandDictWithExtras() in this module.

exception `MiscUtils.DictForArgs.`**DictForArgsError**

Bases: `Exception`

Error when building dictionary from arguments.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

MiscUtils.DictForArgs.**ExpandDictWithExtras**(*d*, *key='Extras'*, *delKey=True*, *dictForArgs=<function dictForArgs>*)

Return a dictionary with the 'Extras' column expanded by dictForArgs().

For example, given:

```
{'Name': 'foo', 'Extras': 'x=1 y=2'}
```

The return value is:

```
{'Name': 'foo', 'x': '1', 'y': '2'}
```

The key argument controls what key in the dictionary is used to hold the extra arguments. The delKey argument controls whether that key and its corresponding value are retained. The same dictionary may be returned if there is no extras key. The most typical use of this function is to pass a row from a DataTable that was initialized from a CSV file (e.g., a spreadsheet or tabular file). FormKit and MiddleKit both use CSV files and allow for an Extras column to specify attributes that occur infrequently.

MiscUtils.DictForArgs.**PyDictForArgs**(*s*)

Build dictionary from arguments.

Takes an input such as:

```
x=3
name="foo"
first='john'; last='doe'
list=[1, 2, 3]; name='foo'
```

And returns a dictionary representing the same.

All values are interpreted as Python expressions. Any error in these expressions will raise the appropriate Python exception. This syntax allows much more power than dictForArgs() since you can include lists, dictionaries, actual ints and floats, etc.

This could also open the door to hacking your software if the input comes from a tainted source such as an HTML form or an unprotected configuration file.

Returns { } for an empty string.

See also: dictForArgs() and expandDictWithExtras() in this module.

MiscUtils.DictForArgs.**dictForArgs**(*s*)

Build dictionary from arguments.

Takes an input such as:

```
x=3
name="foo"
first='john' last='doe'
required border=3
```

And returns a dictionary representing the same. For keys that aren't given an explicit value (such as 'required' above), the value is '1'.

All values are interpreted as strings. If you want ints and floats, you'll have to convert them yourself.

This syntax is equivalent to what you find in HTML and close to other ML languages such as XML.

Returns { } for an empty string.

The informal grammar is:

```
(NAME [=NAME|STRING])*
```

Will raise DictForArgsError if the string is invalid.

See also: pyDictForArgs() and expandDictWithExtras() in this module.

`MiscUtils.DictForArgs.`**`expandDictWithExtras`**`(`*d*, *key='Extras'*, *delKey=True*, *dictForArgs=<function dictForArgs>*`)`

Return a dictionary with the 'Extras' column expanded by dictForArgs().

For example, given:

```
{'Name': 'foo', 'Extras': 'x=1 y=2'}
```

The return value is:

```
{'Name': 'foo', 'x': '1', 'y': '2'}
```

The key argument controls what key in the dictionary is used to hold the extra arguments. The delKey argument controls whether that key and its corresponding value are retained. The same dictionary may be returned if there is no extras key. The most typical use of this function is to pass a row from a DataTable that was initialized from a CSV file (e.g., a spreadsheet or tabular file). FormKit and MiddleKit both use CSV files and allow for an Extras column to specify attributes that occur infrequently.

`MiscUtils.DictForArgs.`**`pyDictForArgs`**`(`*s*`)`

Build dictionary from arguments.

Takes an input such as:

```
x=3
name="foo"
first='john'; last='doe'
list=[1, 2, 3]; name='foo'
```

And returns a dictionary representing the same.

All values are interpreted as Python expressions. Any error in these expressions will raise the appropriate Python exception. This syntax allows much more power than dictForArgs() since you can include lists, dictionaries, actual ints and floats, etc.

This could also open the door to hacking your software if the input comes from a tainted source such as an HTML form or an unprotected configuration file.

Returns { } for an empty string.

See also: dictForArgs() and expandDictWithExtras() in this module.

19.6.9 Error

Universal error class.

class `MiscUtils.Error.`**`Error`**(*obj*, *message*, *valueDict=None*, ***valueArgs*)

Bases: `dict`

Universal error class.

An error is a dictionary-like object, containing a specific user-readable error message and an object associated with it. Since Error inherits dict, other informative values can be arbitrarily attached to errors. For this reason, subclassing Error is rare.

Example:

```
err = Error(user, 'Invalid password.')
err['time'] = time.time()
err['attempts'] = attempts
```

The object and message can be accessed via methods:

```
print(err.object())
print(err.message())
```

When creating errors, you can pass None for both object and message. You can also pass additional values, which are then included in the error:

```
>>> err = Error(None, 'Too bad.', timestamp=time.time())
>>> err.keys()
['timestamp']
```

Or include the values as a dictionary, instead of keyword arguments:

```
>>> info = {'timestamp': time.time()}
>>> err = Error(None, 'Too bad.', info)
```

Or you could even do both if you needed to.

`__init__`(*obj*, *message*, *valueDict=None*, ***valueArgs*)

Initialize the error.

Takes the object the error occurred for, and the user-readable error message. The message should be self sufficient such that if printed by itself, the user would understand it.

`message`()

Get the user-readable error message.

`object`()

Get the object the error occurred for.

19.6.10 Funcs

MiscUtils.Funcs

This module holds functions that don't fit in anywhere else.

You can safely import * from MiscUtils.Funcs if you like.

`MiscUtils.Funcs.`**`asclocaltime`**`(`*t=None*`)`

Return a readable string of the current, local time.

Useful for time stamps in log files.

`MiscUtils.Funcs.`**`charWrap`**`(`*s*, *width*, *hanging=0*`)`

Word wrap a string.

Return a new version of the string word wrapped with the given width and hanging indent. The font is assumed to be monospaced.

This can be useful for including text between `<pre>...</pre>` tags, since `<pre>` will not word wrap, and for lengthy lines, will increase the width of a web page.

It can also be used to help delineate the entries in log-style output by passing hanging=4.

`MiscUtils.Funcs.`**`commas`**`(`*number*`)`

Insert commas in a number.

Return the given number as a string with commas to separate the thousands positions.

The number can be a float, int, long or string. Returns None for None.

`MiscUtils.Funcs.`**`excstr`**`(`*e*`)`

Return a string for the exception.

The string will be in the format that Python normally outputs in interactive shells and such:

```
<ExceptionName>: <message>
AttributeError: 'object' object has no attribute 'bar'
```

Neither str(e) nor repr(e) do that.

`MiscUtils.Funcs.`**`hostName`**`()`

Return the host name.

The name is taken first from the os environment and failing that, from the 'hostname' executable. May return None if neither attempt succeeded. The environment keys checked are HOST and HOSTNAME, both upper and lower case.

`MiscUtils.Funcs.`**`localIP`**`(`*remote=('www.yahoo.com', 80)*, *useCache=True*`)`

Get the "public" address of the local machine.

This is the address which is connected to the general Internet.

This function connects to a remote HTTP server the first time it is invoked (or every time it is invoked with useCache=0). If that is not acceptable, pass remote=None, but be warned that the result is less likely to be externally visible.

Getting your local ip is actually quite complex. If this function is not serving your needs then you probably need to think deeply about what you really want and how your network is really set up. Search comp.lang.python for "local ip" for more information.

`MiscUtils.Funcs.`**`localTimeDelta`**`(`*t=None*`)`

Return timedelta of local zone from GMT.

`MiscUtils.Funcs.`**`positiveId`**`(`*obj*`)`

Return id(obj) as a non-negative integer.

`MiscUtils.Funcs.`**`safeDescription`**`(`*obj*, *what='what'*`)`

Return the repr() of obj and its class (or type) for help in debugging.

A major benefit here is that exceptions from repr() are consumed. This is important in places like "assert" where you don't want to lose the assertion exception in your attempt to get more information.

Example use:

```
assert isinstance(foo, Foo), safeDescription(foo)
print("foo:", safeDescription(foo))  # won't raise exceptions

# better output format:
assert isinstance(foo, Foo), safeDescription(foo, 'foo')
print(safeDescription(foo, 'foo'))
```

`MiscUtils.Funcs.`**`timestamp`**`(`*t=None*`)`

Return a dictionary whose keys give different versions of the timestamp.

The dictionary will contain the following timestamp versions:

```
'tuple': (year, month, day, hour, min, sec)
'pretty': 'YYYY-MM-DD HH:MM:SS'
'condensed': 'YYYYMMDDHHMMSS'
'dashed': 'YYYY-MM-DD-HH-MM-SS'
```

The focus is on the year, month, day, hour and second, with no additional information such as timezone or day of year. This form of timestamp is often ideal for print statements, logs and filenames. If the current number of seconds is not passed, then the current time is taken. The 'pretty' format is ideal for print statements, while the 'condensed' and 'dashed' formats are generally more appropriate for filenames.

`MiscUtils.Funcs.`**`uniqueId`**`(`*forObject=None*`)`

Generate an opaque identifier string made of 32 hex digits.

The string is practically guaranteed to be unique for each call.

If a randomness source is not found in the operating system, this function will use SHA-256 hashing with a combination of pseudo-random numbers and time values to generate additional randomness. In this case, if an object is passed, then its id() will be incorporated into the generation as well.

`MiscUtils.Funcs.`**`valueForString`**`(`*s*`)`

Return value for a string.

For a given string, returns the most appropriate Pythonic value such as None, a long, an int, a list, etc. If none of those make sense, then returns the string as-is.

"None", "True" and "False" are case-insensitive because there is already too much case sensitivity in computing, damn it!

`MiscUtils.Funcs.`**`wordWrap`**`(`*s*, *width=78*`)`

Return a version of the string word wrapped to the given width.

19.6.11 MixIn

`MiscUtils.MixIn.`**`MixIn`**(*pyClass*, *mixInClass*, *makeAncestor=False*, *mixInSuperMethods=False*)

Mixes in the attributes of the mixInClass into the pyClass.

These attributes are typically methods (but don't have to be). Note that private attributes, denoted by a double underscore, are not mixed in. Collisions are resolved by the mixInClass' attribute overwriting the pyClass'. This gives mix-ins the power to override the behavior of the pyClass.

After using MixIn(), instances of the pyClass will respond to the messages of the mixInClass.

An assertion fails if you try to mix in a class with itself.

The pyClass will be given a new attribute mixInsForCLASSNAME which is a list of all mixInClass' that have ever been installed, in the order they were installed. You may find this useful for inspection and debugging.

You are advised to install your mix-ins at the start up of your program, prior to the creation of any objects. This approach will result in less headaches. But like most things in Python, you're free to do whatever you're willing to live with. :-)

There is a bitchin' article in the Linux Journal, April 2001, "Using Mix-ins with Python" by Chuck Esterbrook, which gives a thorough treatment of this topic.

An example, that resides in the Webware MiddleKit plug-in, is MiddleKit.Core.ModelUser.py, which install mix-ins for SQL interfaces. Search for "MixIn(".

If makeAncestor is True, then a different technique is employed: a new class is created and returned that is the same as the given pyClass, but will have the mixInClass added as its first base class. Note that this is different from the behavior in legacy Webware versions, where the __bases__ attribute of the pyClass was changed. You probably don't need to use this and if you do, be aware that your mix-in can no longer override attributes/methods in pyClass.

If mixInSuperMethods is True, then support will be enabled for you to be able to call the original or "parent" method from the mixed-in method. This is done like so:

```
class MyMixInClass:
def foo(self):
    MyMixInClass.mixInSuperFoo(self)  # call the original method
    # now do whatever you want
```

19.6.12 NamedValueAccess

NamedValueAccess.py

NamedValueAccess provides functions for accessing Python objects by keys and named attributes. A 'key' is a single identifier such as 'foo'. A 'name' could be a key, or a qualified key, such as 'foo.bar.boo'. Names are generally more convenient and powerful, while the key-oriented function is more efficient and provide the atomic functionality that the name-oriented function is built upon.

CREDIT

Chuck Esterbrook <echuck@mindspring.com> Tavis Rudd <tavis@calrudd.com>

exception `MiscUtils.NamedValueAccess.`**`NamedValueAccessError`**

Bases: `LookupError`

General named value access error.

`__init__`(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception MiscUtils.NamedValueAccess.**ValueForKeyError**

Bases: *NamedValueAccessError*

No value for key found error.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

MiscUtils.NamedValueAccess.**valueForKey**(*obj*, *key*, *default=<class 'MiscUtils.NoDefault'>*)

Get the value of the object named by the given key.

This method returns the value with the following precedence:

1. Methods before non-methods
2. Attributes before keys (__getitem__)
3. Public things before private things (private being denoted by a preceding underscore)

Suppose key is 'foo', then this method returns one of these:

- obj.foo()
- obj._foo()
- obj.foo
- obj._foo
- obj['foo']
- default # only if specified

If all of these fail, a ValueForKeyError is raised.

NOTES

- valueForKey() works on dictionaries and dictionary-like objects.
- See valueForName() which is a more advanced version of this function that allows multiple, qualified keys.

MiscUtils.NamedValueAccess.**valueForName**(*obj*, *name*, *default=<class 'MiscUtils.NoDefault'>*)

Get the value of the object that is named.

The name can use dotted notation to traverse through a network/graph of objects. Since this function relies on valueForKey() for each individual component of the name, you should be familiar with the semantics of that notation.

Example: valueForName(obj, 'department.manager.salary')

19.6.13 ParamFactory

ParamFactory.py

A factory for creating cached, parametrized class instances.

`class MiscUtils.ParamFactory.ParamFactory(klass, **extraMethods)`

Bases: `object`

`__init__(klass, **extraMethods)`

`allInstances()`

19.6.14 PickleCache

PickleCache.py

PickleCache provides tools for keeping fast-loading cached versions of files so that subsequent loads are faster. This is similar to how Python silently caches .pyc files next to .py files.

The typical scenario is that you have a type of text file that gets "translated" to Pythonic data (dictionaries, tuples, instances, ints, etc.). By caching the Python data on disk in pickle format, you can avoid the expensive translation on subsequent reads of the file.

Two real life cases are MiscUtils.DataTable, which loads and represents comma-separated files, and the separate MiddleKit plug-in which has an object model file. So for examples on using this module, load up the following files and search for "Pickle":

```
MiscUtils/DataTable.py
MiddleKit/Core/Model.py
```

The cached file is named the same as the original file with '.pickle.cache' suffixed. The utility of '.pickle' is to denote the file format and the utility of '.cache' is to provide `*.cache` as a simple pattern that can be removed, ignored by backup scripts, etc.

The treatment of the cached file is silent and friendly just like Python's approach to .pyc files. If it cannot be read or written for various reasons (cache is out of date, permissions are bad, wrong python version, etc.), then it will be silently ignored.

GRANULARITY

In constructing the test suite, I discovered that if the source file is newly written less than 1 second after the cached file, then the fact that the source file is newer will not be detected and the cache will still be used. I believe this is a limitation of the granularity of os.path.getmtime(). If anyone knows of a more granular solution, please let me know.

This would only be a problem in programmatic situations where the source file was rapidly being written and read. I think that's fairly rare.

SEE ALSO

https://docs.python.org/3/library/pickle.html

`class MiscUtils.PickleCache.PickleCache`

Bases: `object`

Abstract base class for PickleCacheReader and PickleCacheWriter.

`picklePath(filename)`

class MiscUtils.PickleCache.**PickleCacheReader**

Bases: *PickleCache*

picklePath(*filename*)

read(*filename*, *pickleProtocol=None*, *source=None*, *verbose=None*)

Read data from pickle cache.

Returns the data from the pickle cache version of the filename, if it can read. Otherwise returns None, which also indicates that writePickleCache() should be subsequently called after the original file is read.

class MiscUtils.PickleCache.**PickleCacheWriter**

Bases: *PickleCache*

picklePath(*filename*)

write(*data*, *filename*, *pickleProtocol=None*, *source=None*, *verbose=None*)

MiscUtils.PickleCache.**readPickleCache**(*filename*, *pickleProtocol=None*, *source=None*, *verbose=None*)

Read data from pickle cache.

Returns the data from the pickle cache version of the filename, if it can read. Otherwise returns None, which also indicates that writePickleCache() should be subsequently called after the original file is read.

MiscUtils.PickleCache.**writePickleCache**(*data*, *filename*, *pickleProtocol=None*, *source=None*, *verbose=None*)

19.6.15 PickleRPC

PickleRPC.py

PickleRPC provides a Server object for connection to Pickle-RPC servers for the purpose of making requests and receiving the responses.

```
>>> from MiscUtils.PickleRPC import Server
>>> server = Server('http://localhost:8080/Examples/PickleRPCExample')
>>> server.multiply(10,20)
200
>>> server.add(10,20)
30
```

See also: Server, PickleRPCServlet, Examples.PickleRPCExample

UNDER THE HOOD

Requests look like this:

```
{
    'version':    1,  # default
    'action':     'call',  # default
    'methodName': 'NAME',
    'args':       (A, B, ...),  # default = (,)
    'keywords':   {'A': A, 'B': B, ...}  # default = {}
}
```

Only ‘methodName’ is required since that is the only key without a default value.

Responses look like this:

```
{
    'timeReceived': N,
    'timeReponded': M,
    'value': V,
    'exception': E,
    'requestError': E,
}
```

'timeReceived' is the time the initial request was received. 'timeResponded' is the time at which the response was finished, as close to transmission as possible. The times are expressed as number of seconds since the Epoch, e.g., `time.time()`.

Value is whatever the method happened to return.

Exception may be 'occurred' to indicate that an exception occurred, the specific exception, such as "KeyError: foo" or the entire traceback (as a string), at the discretion of the server. It will always be a non-empty string if it is present.

RequestError is an exception such as "Missing method in request." (with no traceback) that indicates a problem with the actual request received by the Pickle-RPC server.

Value, exception and requestError are all exclusive to each other.

SECURITY

Pickle RPC uses the SafeUnpickler class (in this module) to prevent unpickling of unauthorized classes. By default, it doesn't allow _any_ classes to be unpickled. You can override *allowedGlobals()* or *findGlobal()* in a subclass as needed to allow specific class instances to be unpickled.

Note that both *Transport* in this package and *PickleRPCServlet* in the Webware main package are derived from *Safe-Unpickler*.

CREDIT

The implementation of this module was taken directly from Python's xmlrpclib and then transformed from XML-orientation to Pickle-orientation.

The zlib compression was adapted from code by Skip Montanaro that I found here: http://manatee.mojam.com/~skip/python/

exception `MiscUtils.PickleRPC.`**`Error`**

Bases: `Exception`

The abstract exception/error class for all PickleRPC errors.

`__init__`(**args*, ***kwargs*)

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `MiscUtils.PickleRPC.`**`InvalidContentTypeError`**(*headers*, *content*)

Bases: *`ResponseError`*

Invalid content type error.

`__init__`(*headers*, *content*)

`args`

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception MiscUtils.PickleRPC.**ProtocolError**(*url*, *errcode*, *errmsg*, *headers*)

Bases: *ResponseError*, ProtocolError

RPC protocol error.

__init__(*url*, *errcode*, *errmsg*, *headers*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception MiscUtils.PickleRPC.**RequestError**

Bases: *Error*

Errors originally raised by the server caused by malformed requests.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception MiscUtils.PickleRPC.**ResponseError**

Bases: *Error*

Unhandled exceptions raised when the server was computing a response.

These will indicate errors such as:

- exception in the actual target method on the server
- malformed responses
- non “200 OK” status code responses

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class MiscUtils.PickleRPC.**SafeTransport**

Bases: *Transport*

Handle an HTTPS transaction to a Pickle-RPC server.

allowedGlobals()

Allowed class names.

Must return a list of (moduleName, klassName) tuples for all classes that you want to allow to be unpickled.

Example:

```
return [('datetime', 'date')]
```

Allows datetime.date instances to be unpickled.

findGlobal(*module*, *klass*)

Find class name.

load(*file*)

Unpickle a file.

loads(*s*)

Unpickle a string.

make_connection(*host*, *port=None*, *key_file=None*, *cert_file=None*)

Create an HTTPS connection object from a host descriptor.

parse_response(*f*)

Read response from input file and parse it.

parse_response_gzip(*f*)

Read response from input file, decompress it, and parse it.

request(*host*, *handler*, *request_body*, *verbose=False*, *binary=False*, *compressed=False*, *acceptCompressedResponse=False*)

Issue a Pickle-RPC request.

send_content(*connection*, *request_body*, *binary=False*, *compressed=False*, *acceptCompressedResponse=False*)

Send content.

send_host(*connection*, *host*)

Send host header.

send_request(*connection*, *handler*, *request_body*)

Send request.

send_user_agent(*connection*)

Send user-agent header.

user_agent = 'PickleRPC/1 (Webware for Python)'

class MiscUtils.PickleRPC.**SafeUnpickler**

Bases: `object`

Safe unpickler.

For security reasons, we don't want to allow just anyone to unpickle anything. That can cause arbitrary code to be executed. So this SafeUnpickler base class is used to control what can be unpickled. By default it doesn't let you unpickle any class instances at all, but you can create subclass that overrides allowedGlobals().

Note that the PickleRPCServlet class in the Webware package is derived from this class and uses its load() and loads() methods to do all unpickling.

allowedGlobals()

Allowed class names.

Must return a list of (moduleName, klassName) tuples for all classes that you want to allow to be unpickled.

Example:

```
return [('datetime', 'date')]
```

Allows datetime.date instances to be unpickled.

findGlobal(*module*, *klass*)

Find class name.

load(*file*)

Unpickle a file.

loads(*s*)

Unpickle a string.

class MiscUtils.PickleRPC.**Server**(*uri*, *transport=None*, *verbose=False*, *binary=True*, *compressRequest=True*, *acceptCompressedResponse=True*)

Bases: `object`

uri [,options] -> a logical connection to an XML-RPC server

uri is the connection point on the server, given as scheme://host/target.

The standard implementation always supports the "http" scheme. If SSL socket support is available, it also supports "https".

If the target part and the slash preceding it are both omitted, "/PickleRPC" is assumed.

See the module doc string for more information.

__init__(*uri*, *transport=None*, *verbose=False*, *binary=True*, *compressRequest=True*, *acceptCompressedResponse=True*)

Establish a "logical" server connection.

MiscUtils.PickleRPC.**ServerProxy**

alias of *Server*

class MiscUtils.PickleRPC.**Transport**

Bases: *SafeUnpickler*

Handle an HTTP transaction to a Pickle-RPC server.

allowedGlobals()

Allowed class names.

Must return a list of (moduleName, klassName) tuples for all classes that you want to allow to be unpickled.

Example:

```
return [('datetime', 'date')]
```

Allows datetime.date instances to be unpickled.

findGlobal(*module*, *klass*)

Find class name.

load(*file*)

Unpickle a file.

loads(*s*)

Unpickle a string.

make_connection(*host*, *port=None*)

Create an HTTP connection object from a host descriptor.

parse_response(*f*)

Read response from input file and parse it.

`parse_response_gzip`(*f*)

Read response from input file, decompress it, and parse it.

`request`(*host*, *handler*, *request_body*, *verbose=False*, *binary=False*, *compressed=False*, *acceptCompressedResponse=False*)

Issue a Pickle-RPC request.

`send_content`(*connection*, *request_body*, *binary=False*, *compressed=False*, *acceptCompressedResponse=False*)

Send content.

`send_host`(*connection*, *host*)

Send host header.

`send_request`(*connection*, *handler*, *request_body*)

Send request.

`send_user_agent`(*connection*)

Send user-agent header.

`user_agent = 'PickleRPC/1 (Webware for Python)'`

CHAPTER TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX