WebGUI Developers Guide

by JT Smith
Frank Dillon
Doug Bell
Graham Knop
Chris Nehren
Colin Kuskie

Printed in the United States of America by Lulu.com.

Published by Plain Black Corporation, 1360 Regent St, #145, Madison, WI, 53715

Editor: Kristi McCombs
JT Smith

Cover Design: Darci Gibson

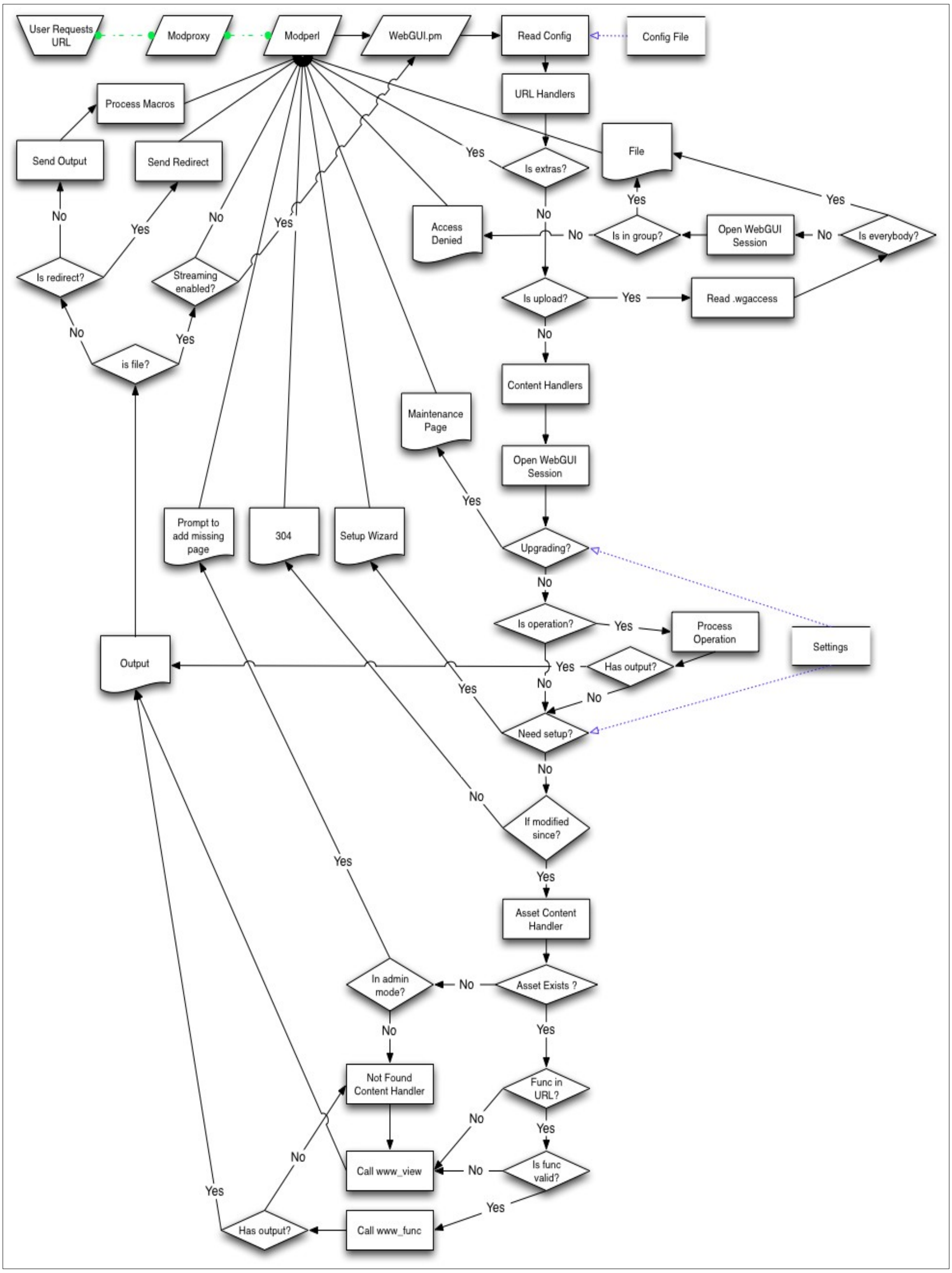
Printing History: July 2008: First Edition
WebGUI Version 7.5

Preface

This version of the *WebGUI Developers Guide* is intended for developers working in WebGUI 7.5 or above. It covers everything necessary to take full advantage of WebGUI's functionality and flexibility. The best of WebGUI's development team has come together to share the inner workings of WebGUI, and provide expert lessons on how to use it to your advantage. Learn about writing custom workflow activities, writing tests, using the WebGUI API, writing and installing your own assets, and a great deal more.

Table of Contents

WebGUI Request Cycle



The WebGUI request cycle is the path that a user request from the browser takes through WebGUI's internals. In general, the path goes from the browser to mod_proxy, to mod_perl, to a URL handler, then a content handler, then an asset, and back to mod_perl, mod_proxy, and is displayed in the user's browser.

mod_proxy

The standard way of running WebGUI is behind a reverse proxy. Because of the large amount of code used by WebGUI, the mod_perl server takes up a significant amount of memory. Using a smaller process to handle the simpler request, such as file access, allows the system to run much faster. This isn't a required part of WebGUI, however, and can be configured by the individual system administrator.

mod_perl

From the reverse proxy, the requests for dynamic content (the most important part of WebGUI) are forwarded to a server using mod_perl. mod_perl is the Perl interface to Apache©, and allows complete control over how Apache handles these requests. This is where all of WebGUI's code lives and is the essential part of the system. Based on the Apache configuration directives (usually virtual host sections), a WebGUI base directory and configuration file is chosen for the request. WebGUI is registered as an Init handler, where it will set itself as a handler for other phases of Apache's request processing.

```
SetHandler perl-script                          # Tell Apache we are using perl
PerlInitHandler WebGUI                          # Register WebGUI for Init
PerlSetVar WebguiRoot /data/WebGUI              # List the location of the WebGUI code
                                                    and config
PerlSetVar WebguiConfig www.example.com.conf# List the configuration file to use
```

WebGUI

The configuration file listed is the first main area used to control WebGUI. It contains essential information such as the database to connect to, the macros to use, and the assets to make available. When developing add-ons for WebGUI, adding it to the configuration file is usually one of the steps. The WebGUI configuration file is also where you'll find the list of URL handlers to be called.

URL Handlers

URL handlers allow you to plug functionality into WebGUI that will handle all functions on a given URL or group of URL's. See the URL Handlers chapter for additional details.

Extras

The first URL handler is the PassThru handler which deals with the /extras folder, where WebGUI stores sharable static content like images and javascript files that are used throughout the system. Three other pieces of information included in the configuration file are the extras, passthrough, and uploads locations. Extras include the major javascript libraries used by WebGUI, such as YUI© and TinyMCE©. These are just static files on the filesystem, so WebGUI directs Apache© to serve them as such. Passthrough URL's are handled the same way, but don't have any particular meaning to WebGUI. These could be other areas with static files, or any other type of handler configured in Apache©. When a proxy server is in use, it would usually be configured to handle these locations to decrease the load on the server.

Uploads

Uploads are another area with static files, but are managed by WebGUI. Often they are set up to be served by the proxy server; however, this places limits on them. WebGUI attaches security information to uploads, but if the handling of these files is short circuited by the proxy server anyone would be able to access them. Uploads are stored in randomly generated path names which provides some obscurity, but for truly sensitive data real permissions handling is needed. Of course, this extra handling isn't without a price. The extra handling to enforce the permissions imposes significant slowdowns.

To handle security on uploads, the ID of the group with access to the file is stored in a file named .wgaccess, which is stored in the same location as the file. Upon retrieving the group ID, WebGUI first checks if it is the Everyone group. If so, it instructs Apache to serve these files. For any other group, though, it has to find the group membership information, which means creating a WebGUI session to read the information out of the database. The session will also determine the currently logged in user, which is then compared to the group listed in the access file. From here, either the file or a forbidden error is returned to the user.

Content

The Content URL handler is another plug-in to the URL handlers system. Content handlers are easier to write, and instead of working from specific URL's, they can come into play by cookies, URL's, form parameters, or any number of other things. The next section in this chapter talks more about the content handlers that come with WebGUI, and you can learn more about them in the Content Handlers chapter.

Content Handlers

After handling as many cases of static files as possible, a database connection is needed for any further processing, and this is where content handlers come in. The next step is to initialize a WebGUI session. Creating the session uses the database information from the configuration file and the Apache request object. From this it determines if the user has an active browser session and retrieves WebGUI's general settings.

Upgrade

As part of WebGUI's upgrade process, it saves a setting in the database marking the site as under maintenance. If this has been set, it will stop processing and display a maintenance screen to users. This is to prevent the code from operating on a different database structure than it was designed for, resulting in possible errors or even data corruption.

Operations

Operations are the first section where major work is done. These are used for anything in WebGUI that doesn't pertain to an Asset. They are called by the client by including op=<operation name> in the query string. The majority of operations are administrative functions, such as the User and Group management systems. The workflow and scheduler system works through operations as well, sending and receiving information from Spectre. Authentication and user profile editing is also done through operations. Operations work through a fixed list defined in the WebGUI::Operation module. This list links each operation name with the perl module that has the code for it. For example, the line:

```
'editGroup' => 'WebGUI::Operation::Group',
```

This line defines that a URL request ending with ?op=editGroup will be handled by the WebGUI::Operation::Group module. It will call the sub www_editGroup, giving the session object as the first parameter. If needed, the module will automatically be loaded. After doing whatever processing is needed, the operation can return output to the client. Alternately, it can return nothing and WebGUI will continue processing the request as if there had not been an operation.

Setup

Setup mode is another special state in which WebGUI can be. This is set when initially creating a WebGUI site. When in setup mode, the client is shown a screen allowing them to set the website's name and define the administrator account. It also includes the Site Starter, where a simple custom theme can be defined. Setup mode will automatically be turned off when this is complete.

Assets

Next is the most important processing area in WebGUI: Assets. Each asset on the site is addressed by a unique URL. WebGUI will use this URL to find and instantiate the asset. If the URL doesn't match any of the assets in the database, WebGUI will do one of two things. If the user is currently in admin mode, s/he will be prompted to create a new asset if desired. Otherwise, WebGUI continues processing, but uses the asset listed in the Admin Settings screen as the “Not Found” page.

Once it has determined that the asset exists, it checks the HTTP cache headers. These headers include the last date and time the client checked for content at a given URL. If the most recent revision of the asset is older than the date sent by the client, there is no need to re-send the same content to the client, so WebGUI returns an HTTP 304 Content Not Modified message.

At this stage, the asset has been determined and it is known that the client needs to be sent the content. WebGUI will check if an alternate entry point has been specified using the func query parameter. If a function has been specified, the www_func method will be called to generate the output. If not, or if the function is otherwise invalid, www_view will be used. This method is responsible for generating the output for the asset. If a function was specified, but didn't output anything when called, it will again fall back on the www_view method.

Output

The asset or operation has several options for handling its output. The most common method is to use the session's output object. With this, the asset outputs content directly to the client, and then needs to return the magic value “chunked”, indicating that the output has been delivered. This allows the asset to output larger amounts of data without loading it all into memory at once. The output object will also process macros. Although specific assets use the macro API to process them earlier, macros are generally processed immediately before delivering the content to the client. If WebGUI finds the value “chunked,” it stops processing the request, as there is no more work to be done.

The next option the asset has for output is to point to a static file on the filesystem. In this case, it tells the session what file it wants to output. If WebGUI is configured to enable streaming these files, it will tell Apache the location of the file and instruct it to serve it.

The last option available for output is a redirect. Similar to a file, the asset needs to tell the WebGUI session it wants to perform a redirect. When getting to this stage, it will output the HTTP header needed to direct the client to the new URL.

If the asset hasn't specified any of these special cases, it is expected to return the full data to send to the client. That data is then sent to the client, which is the last stage of the request cycle.

Testing WebGUI

Most likely, you're reading this book because you're a developer - a hacker. You are gifted in logical thinking, problem solving, caffeine consumption, computer programming, handling sleep deprivation, system administration, typing fast and drooling over the latest in hardware. The thought of meetings, dealing with users (or managers) and writing documentation are probably not high on your list of priorities. Testing your software falls in the same category, but past experience has shown that not testing can be severely limiting in your career. The good news is that there is a way to make software testing easier, and even fun.

Testing has usually been done by hand and involves installing the software, resetting the database, restarting Apache, firing up a browser, logging in, turning on admin mode, adding and configuring the asset, committing the asset, adding users and groups for permissions, checking all the screens and options... are you frustrated just thinking about it? And if that weren't enough, with each change to your code, you have to restart that whole process again. It is also a process fraught with problems, because if you forget a step, you might miss a bug.

Automated software testing helps makes the process easier, by allowing you to do all those things with code. That's right. You write code to test your code. This is actually a good thing, because you're good at writing code. You write a test once, and then run it again and again, and it will do the same thing every time. No more clicks, no more wondering about whether you did things in the right order or dropped a step.

Your tests can also function as a kind of documentation, since other developers can read your tests and see how you expect your code to be used. If bugs are found, then you can write tests to duplicate the bug and make sure that it gets fixed, and perhaps more importantly, that it stays fixed in the future as libraries change, or other people maintain your code. You can write the tests before you write your code. When all the tests pass, you're done!

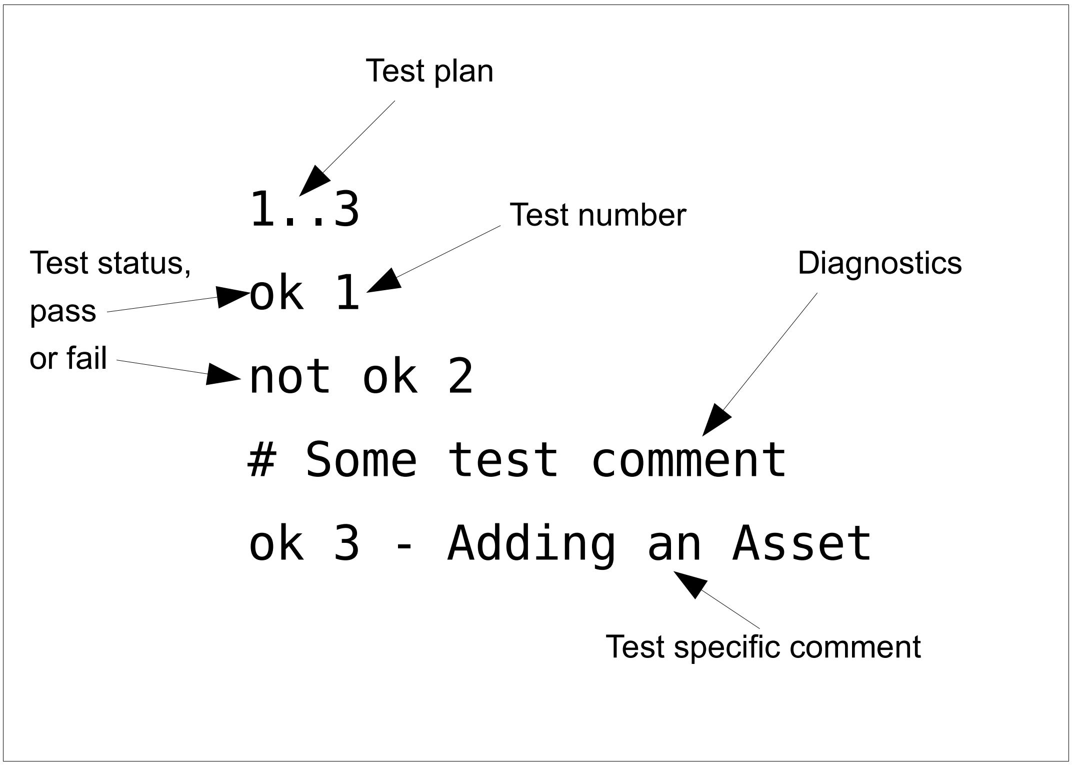
This is called “test driven development, and it is a part of Extreme Programming.

Hopefully, this has whet your appetite and raised your interest in automated testing. This chapter is devoted to showing you how it's done.

Writing Tests in Perl

WebGUI is written in Perl, and all of its tests are written in Perl. There is a whole set of testing modules in Perl which provide methods for simple, scalar tests, testing data structures, simplified browser testing and even automating the writing of tests. These different modules all emit output using a standard called TAP (Test Anything Protocol).

In the future, to test AJAX and other functions, some of the tests may be written in Javascript using Selenium© or some similar browser-based automated testing framework.



```
1..3
ok 1
not ok 2
# Some test comment
ok 3 - Adding an Asset
```

The example above shows a sample TAP output. TAP is a very simple format with five main parts:

1. Test Plan: At the beginning is the number of tests that are expected to be run. This is set in the test script, and helps any program that post processes the output to know if the test died before all tests were run.

2. Test Status: Each test outputs a simple ok or not ok message, depending on whether the test passed or failed.

3. Test Number: The testing module automatically takes care of numbering the tests. Since comments in tests are not required by the format, this can help to identify tests that are failing.

4. Test Specific Comments: Optional test specific comments can give details about each test, such as what is being tested. This helps a lot in debugging tests that fail, and I strongly encourage ("strongly encourage" were the words my Dad used to say when he meant, "Do this or you'll get whupped.") you to write descriptive comments for each and every test that you write.

5. Diagnostics: Diagnostics can be created by the test to show why it failed: "Expected to get an Asset, but got back a User object instead," or added by the test script to show sections, debugging output, or anything else.

TAP is a very human readable format, but going through the output of hundreds of test scripts, where each script has 50 or so tests, is neither easy nor fun. So, in addition to testing modules to help you generate TAP, there are test harnesses which take care of running sets of test scripts and summarizing their output.

```
yourTest...... Failed 1/3 subtests

Test Summary Report
foo.t (Wstat: 0 Tests: 3 Failed: 1)
  Failed test number(s):  2
Files=1, Tests=3,  0 wallclock secs ( 0.02 usr +  0.01 sys =  0.03 CPU)
Result: FAIL
```

The above example shows the output of a test harness called prove. Depending on the version of the Test::Harness module installed with Perl on your system, this output can be different. Now that you've seen TAP, let's see how to generate it with Test::More.

Test::More

Test::More is the workhorse of Perl testing, providing a whole slew of testing subroutines. However, before you begin using them, you need to generate the test plan. This can be done either when you use the Test::More module, or later using the plan subroutine.

```
use Test::More tests => 5;
OR
my @testData = generatedTests();
plan tests => 15 + scalar @testData;
```

Using the plan subroutine gives you the freedom to calculate the number of tests. For example, you may need to test each file in a directory, or you may have data driven tests as shown above. Regardless of which method you choose, you're now ready to start writing tests.

The most basic test method is called ok.

```
ok( $test, 'Test passed');
```

If $test is true (as Perl defines true), then the test prints "ok". Otherwise it prints "not ok". In both cases, the test output is automatically numbered for you, and the comment 'Test passed' is appended to the output.

You can test everything with ok. All that you need to do is perform the comparison yourself and then test the result of the comparison. To test for falseness, just invert the variable.

```
my @sons = $family->getChildren;
my $hobbyCheck = $sons->[1]->getHobby eq 'Trains';
ok($hobbyCheck, 'Second son still likes trains');
my $match = ($home =~ /kids/);
ok($match, 'The kids are in $home');
ok($sons->[0]->bouncy, 'My first son is bouncy');
my $ageDifference = $sons->[0]->getAge - $sons->[1]->getAge - 2;
ok(!$ageDifference, 'Sons are two years apart in age');
```

That works, but it has a few problems:

1. It's awkward. The numeric check for the difference in ages had to be coerced into a boolean check, and then logically inverted to make the test pass.

2. It uses a lot of temporary variables. Now, strictly, they did not have

to be created, but if the test fails, it is a lot easier to print them out when they are in separate variables, rather than "flattened" into the first argument of ok.

3. When the test fails, it just prints 'not ok', and not why, since ok is just a boolean test.

Fortunately, Test::More provides several more testing subroutines that fix those problems.

```
my @sons = $family->getChildren;
is($sons->[1]->getHobby, 'Trains',
        'Second son still likes trains');
like($home->status, qr/kids/, 'The kids are in $home');
ok($sons->[0]->bouncy, 'My first son is bouncy');
cmp_ok(
            $sons->[0]->getAge - $sons->[1]->getAge,
      '==', 2,
      'Sons are two years apart in age'
);
```

Most of your tests will use is and like :

```
is($test, $expected, $comment);
like($test, $regexp, $comment);
```

is does a string comparison between $test and $expected., almost the same as doing

```
ok( $test eq $expected, $comment );
```

except that the diagnostics are better if the test fails.

```
is($youngSon->wantsToEat, $youngSon->askedFor,
   'Young son asked for a hot dog');
not ok 1 -  Young son asked for a hot dog
#   Failed test 'Young son asked for a hot dog'
#   at foo.t line 8.
#          got: 'hot dog'
#     expected: 'pizza'
# Looks like you failed 1 test of 1.
```

If you had used ok instead of is, $youngSon would still be having a fit, but now you understand why.

There is also an inverted form of is, called isn't (for those of you who are developers and students of grammar, isn't will Do What You Mean). isn't passes if the first two arguments are not equal, as strings.

For checking parts of a string or for doing anything more complex than a simple string comparison, you can use regular expressions with like.

```
like($olderSon, qr/$father/,
   'Just a hack off the old block');
not ok 1 - Just a hack off the old block
#   Failed test 'Just a hack off the old block'
#   at foo.t line 10.
#                   'blue-eyed'
#     doesn't match '(?-xism:short-tempered)'
# Looks like you failed 2 tests of 2.
```

The first argument to like is a literal or variable. The second argument is a regular expression that will be matched against the first. If the test fails, as shown above, then the diagnostics will show both the actual text and regular expression that were used in the test, in case they were dynamic, as in the example. As with is, there is a negated version of like called unlike, which passes if the text does not match the regular expression.

cmp_ok is a more general version of is. is always does a string comparison on the two values, checking them for equality. cmp_ok allows you to choose how the comparison takes place, either as strings or numbers.

```
cmp_ok($test, $operator, $expected, $comment);
```

cmp_ok allows you to get around problems with comparing floats versus integers.

```
is('2.00', '2', 'Float vs integer equality in strings');
cmp_ok('2.00', '==', '2', 'Float vs integer equality as numbers');
not ok 3 - Float vs integer equality in strings
#   Failed test 'Float vs integer equality in strings'
#   at foo.t line 12.
#          got: '2.00'
#     expected: '2'
ok 4 - Float vs integer equality as numbers
```

As strings, '2.00' and '2' are quite different, but numerically they are the same. Any of the standard relational operators can be passed to cmp_ok allowing you to say things like "five or more", or "fewer than 2".

```
cmp_ok($wife->trips_taken,      '>=', '5', 'Wife is happy');
cmp_ok($youngSon->timeouts,     '<',  '2', 'Young son is happy');
cmp_ok($olderSon->trains_ridden, '>=', '3', 'Older son is happy');
```

Finally, it's still okay to use ok, but only if you're checking truthfulness or falsehood, and not for specific values. After all, if what you're testing says that it returns false, then it could return 0, the empty string '' or undef, and change what it returns in various versions of the code.

Up to this point, all of the tests shown have been for scalar variables. Lest you think that hashes, arrays and objects have been neglected, it's time to point out Test::More's cmp_deeply. It's good for quick, surface checks of data structures. However, the Test::Deep module provides methods for object method checking, order insensitive array checks, and many, many more tests with better diagnostics than cmp_deeply provides.

Test::More provides several other methods for tests; you should spend time reading the POD for the module. More importantly, Test::More provides ways to manage sets of tests.

As you begin writing tests, you'll find code that causes your test scripts to crash, which prevents any tests from running after that point. For example, the code in the earlier example actually throws an exception a few seconds after returning "pizza". You know that $youngSon should not throw an exception after being given what he asked for, but you don't have time to go and diagnose what's going on right now, so you decide to skip the test:

```
SKIP: {
skip "Avoiding food conflicts", 1 if $menu_has_pizza;
is($youngSon->willEat, $youngSon->askedFor,
   'Young son asked for a hot dog');
}
ok 1 # skip Avoiding food conflicts
```

Skipping a test, or a set of tests consists of two parts. First, you mark the tests that you want to skip by placing them in a named SKIP: block. Inside the SKIP: block, you use the skip subroutine to give a reason why the tests will be skipped, and to define how many tests will be skipped. Skipping can be made optional by using a conditional with the skip statement. In our case, if there's no pizza on the menu, it's safe to run the test since $youngSon won't ask for it after saying he wants a hot dog.

The skipped tests will not be run, thus avoiding any potential problems (so long as $youngSon doesn't decide he'll eat a hamburger instead). Test::More will generate the requested number of lines of TAP output, with the tag skip and the reason for skipping, rather than the comments for each test.

You may also want to handle tests that are failing, and you know that they will fail. For example, consider the first test in. In September, with the fifth trip of the year set to beautiful Madison, Wisconsin in October, I don't want the test to continue to fail. In retrospect, I should have used a different method in the Wife class rather than trips_taken. Something like trips_planned would have been much better, but the Wife class doesn't have that method yet. Until then, the test can be marked as passing, even if it actually fails.

First, you place the test inside of a named TODO: block. Then inside of there you localize the $TODO variable and assign to a string which explains why the tests are marked as TODO. When the test runs, the contents of $TODO will be appended to the test comment and the test will be counted as passing.

```
TODO: {
local $TODO = 'Need to add a trips_planned method to $wife';
cmp_ok($wife->trips_taken,      '>=', '5', 'Wife is happy');
}
not ok 1 - Wife is happy # TODO Need to add a trips_planned method to $wife
#   Failed (TODO) test 'Wife is happy'
#   at foo.t line 26.
#     '4'
#         >=
#     '5'
```

Testing in WebGUI

Now that you've learned methods for writing tests, you need to know how to handle the WebGUI specific parts of testing. The most important thing, as you've seen from the earlier chapters in this book, is that you absolutely must have a session variable. You can't do anything in WebGUI without a session variable (that's not strictly true, since you can call some functions out of WebGUI::Utility, but that's not all that interesting).

A lot of work has been put into making WebGUI testing as easy as possible, and that work is encapsulated into WebGUI::Test. It will create and destroy a session variable for your tests, tell Perl how to find the WebGUI library, and provide convenience methods for getting to several files and directories, such as the WebGUI library and root directories, the testing collateral directory and the WebGUI configuration file that was used to create the session.

To get access to all of that, you need two things:

1. Use the WebGUI::Test module in your tests. If you build your tests starting with the test skeleton, /data/WebGUI/t/_test.skeleton, you'll be set!

2. Set the WEBGUI_CONFIG variable, to the absolute path to the WebGUI config file that you want to be used for your tests.

WebGUI Testing Modules

Included here are a number of WebGUI Testing Modules and some examples to help illustrate them.

WebGUI::Test::Maker::Permissions

This module is designed to make it easy to test user level permissions for web access methods, such as WebGUI::Asset::www_view and WebGUI::Operation::www_addGroupsToGroupSave. Simply create an object, configure it with a session variable, the subroutine or method to call for testing, and lists of users that you expect to pass the test, and to fail them. The object does all the hard work for you.

In the example below, a WebGUI::Test::Maker::Permission object was built, and then set up to test WebGUI::Asset's canAdd method. Admin (userId 3) and a test user in a group will be able to call the method. Visitor (userId 1) and a different test user will not. The object will run eight tests, two for each user. In the first test, the requested user is set as the default user in the session object. The second test tests sending that user's userId as a parameter to the method.

```
my $canAddMaker = WebGUI::Test::Maker::Permission->new();
$canAddMaker->prepare({
    'className' => 'WebGUI::Asset',
    'session'   => $session,
    'method'    => 'canAdd',
    'pass'       => [3, $testUsers{'canAdd group user'} ],
    'fail'      => [1, $testUsers{'regular user'},                            ],
});
$canAddMaker->run;
```

WebGUI::PseudoRequest

This handles almost everything that a real Apache2::Request object does in WebGUI, including managing headers, setting and retrieving the request status, form processing and file uploads, except you don't have to have a httpd server process running to make it work. A WebGUI::PseudoRequest object is put into your session variable automatically when you use WebGUI::Test. This is more convenient than creating one when you need it and allows you to easily test most of WebGUI.

WebGUI::PseudoRequest has one known shortcoming. If the session variable has a valid request object, then some of the WebGUI core code will try to load mod_Perl code for handling cookies or to actually send the HTTP header information. However, this is only encountered infrequently, and having a valid request object outweighs the occasional inconvenience. The way to handle those cases is to set the request object inside the session to undef.

```
$session->request->uploadFiles(
    'oneFile',
    [ WebGUI::Test->getTestCollateralPath('WebGUI.pm') ],
);
is($fileStore->addFileFromFormPost('oneFile'), 'WebGUI.pm', 'Return the name of the
uploaded file');
```

How Good are Your Tests?

Eventually, you'd like to know how effective your tests are. Have you wasted effort in writing too many tests? Where do new tests need to be written? These questions are all answered by analyzing code coverage. In code coverage, some "third-party" tool keeps track of every line of code that gets run during the test and in the end. It tells you:

1. Which subroutines were called and which were not.

2. If all conditionals walked down both the true and the false branches.

3. Whether all pieces of a complex conditional statement were exercised. For example, if your code uses $a || $b, did it test $a true, $b true, and $a and $b true?

In Perl, the "third-party" tool is called Devel::Cover. It analyzes all of the metrics above, as well as tell you if your code has enough POD and gives you an aggregate overall score for your code. Devel::Cover has good documentation, and you can refer to it for details on filtering which code is analyzed and which isn't.

For a quick reference, here's how to run a coverage test across the whole WebGUI test suite.

```
> cover -delete ## To delete old data in the coverage database
> env WEBGUI_CONFIG=/data/WebGUI/etc/mywebgui.conf Perl5OPT='-MDevel::Cover'
prove /data/WebGUI/t
> cover ##to generate the coverage report in HTML format.
```

Use the coverage report in the coverage directory, cover_db/coverage.html, to see which areas still need tests.

However, even if the coverage report shows that a module has 100% coverage, it doesn't mean that there are no bugs in the code. First of all, remember that the quality of coverage is determined by the quality of your tests. If your test has a hole in it, even though it covers the code, there could still be a bug in there. Regular expressions are another place to watch out. As long as you pass one piece of data through the regexp, Devel::Cover will consider it covered. However, the regexp may not be fully exercised. If you have code that divides two numbers, Devel::Cover will report it as covered too, but it won't check to see if you've handled divide by zero errors.

Use Devel::Cover to determine what code needs to be covered more thoroughly, but never forget to write edge and corner tests, too. Your goal should be not only to have 100% coverage of your tests, but to have strong, robust tests as well.

General Advice for Testing

- Never, ever run a test on a production website, unless you have a full back-up of:
  - the database
  - the uploads area
- Never run a test with a config file without having a full back-up of the config file. The tests will modify the config file, and in the process, any comments will be stripped from the file.
- Always start a new test with the test skeleton file, _test.skeleton.
- Always clean up after your tests. Delete any users, groups, assets, version tags, workflow activities, database tables or entries, ads, ad spaces, products, macros or storage locations that you create for tests. If you change a config file or a setting, put it back in its "normal" state at the end of the test in an END block. This always resets the environment into a known state for the next test.
- Some assets need to be committed before you can add children to them. Get in the habit of creating an asset, then committing it, then adding children to it. The children should be contained in their own version tag. Don't forget to rollback all the version tags at the end of your test to clean them all up.
- Every test should have a plan, so that if it dies it can be detected in the test run.
- Use good programming techniques. Don't cop out on code quality just because you're writing a test.
- Choose the right test for the right job. My examples in the beginning of the chapter were corny and contrived, but they showed how using the right test method can make your test more robust and provide verbose diagnostics in the event of a failure.
- Remember that in the Scientific Method that success does not prove your theory. When developing new tests, make sure that they fail when you expect them to. If they don't then the theory that your

tests protect against failure is false!

Other Testing Resources

“Test::Tutorial”
by Michael Schwern and the Perl QA Dancers
An excellent introduction to basic testing.
http://search.cpan.org/search?query=Test%3A%3ATutorial&mode=all

Perl Testing - A Developer's Notebook
Ian Langworth and chromatic
O'Reilly Publishers
The canonical book on Perl testing, covering many different modules and approaches to testing with a very, "How do I get this done?" approach.

Perl Best Practices
Damian Conway
O'Reilly Publishers
Chapter 18 goes much more in depth as to why you should test, and covers some good strategies for testing.

“Devel::Cover”
Paul Johnson
http://search.cpan.org/~pjcj/Devel-Cover

“Test::Deep”
Fergal Daly
http://search.cpan.org/~fdaly/Test-Deep

The WebGUI API

All software needs documentation, and WebGUI is no exception. WebGUI has excellent API documentation available. The point of this chapter isn't to describe how to do things with the WebGUI API, but rather to describe where to find its documentation, how to access that documentation, and how to install Perl modules via CPAN to facilitate custom development work. If you'd prefer browsing the API docs in a web browser, you can access the documentation online or generate the HTML documentation yourself.

The API Docs in HTML

There are two ways to access the WebGUI API docs via HTML, and thus via a web browser. The first is to use the site which Plain Black has set up for this. You can get to the documentation for the version of WebGUI you're using by pointing your browser at http://www.plainblack.com/downloads/builds/major.minor.patch-releaseType/api/ . major.minor.patch corresponds to your WebGUI version. For example, if you're running version 7.4.22-stable, you'd browse to http://www.plainblack.com/downloads/builds/7.4.22-stable/api/ . Conversely, if you're running 7.5.3-beta, you'd go to http://www.plainblack.com/downloads/builds/7.5.3-beta/api/ .

If you'd rather generate the documentation for local / offline viewing, you can do that too. The following Perl program will recurse through the /data/WebGUI directory and create the API docs at /data/docs.

```
#!/usr/bin/Perl

#-------------------------------------------------------------------
# WebGUI is Copyright 2001-2008 Plain Black Corporation.
#-------------------------------------------------------------------
# Please read the legal notices (docs/legal.txt) and the license
# (docs/license.txt) that came with this distribution before using
# this software.
#-------------------------------------------------------------------
# http://www.plainblack.com                     info@plainblack.com
#-------------------------------------------------------------------

# strict and warnings are obligatory
```

```
use strict;
use warnings;

# pull in modules that we use
use Getopt::Long;
use File::Path;
use Pod::Html;

# set defaults that the user can override
my $webGUIRoot      = '/data/WebGUI';
my $pod2html        = '/usr/bin/pod2html';
my $rootDirectory   = '/data/docs';
my $filterContent;

# get those overrides
GetOptions(
        'pod2html=s'    => \$pod2html,
    'webGUIRoot=s'  => \$webGUIRoot,
    'filterContent' => \$filterContent,
    'rootDirectory' => \$rootDirectory
);

my $basedir = "$webGUIRoot/lib/WebGUI";

# start recursing
buildFromDir();

sub buildFromDir {

    # get our current directory. default to nothing on the first call.
    my $dir = shift || "";

    # grab all the files in this directory
    opendir my $currentDirectory, "$basedir/$dir";
    my @files = readdir $currentDirectory;
    closedir $currentDirectory;
    @files = sort @files;

    # need to make the directory in the docs tree if this is the first one
    my $first = 1;
    foreach my $file (@files) {
        if ($file =~ /(.*?)\.pm$/) {
            # first one; make that directory
            if ($first) {
                print "Making API docs directory: $rootDirectory/api/$dir\n";
                mkpath(["$rootDirectory/api/$dir"]);
                $first = 0;
            }
            # finally, generate the output
            my $outfile = "$rootDirectory/api/$dir/$1.html";
            print "Generating docs for $basedir/$dir/$file\n";
            pod2html('--quiet', '--noindex', "--outfile=$outfile",
                "$basedir/$dir/$file");

            # filter it if told to
            filterContent($outfile) if $filterContent;
        }

        # looks like we got a directory, recurse into it
        elsif ($file ne "." && $file ne "..") {
            buildFromDir($dir."/".$file);
        }
    }
}

sub filterContent {

    # get the current file
    my $file = shift;
    print "Filtering content for $file\n";

    # slurp in its contents
    my $content = do {
        open my $fh, '<', $file;
        local $/;
        <$fh>;
    };

    # prettify it
    $content =~ s/NOTE:/<b>NOTE:<\/b>/ig;
    $content =~ s/TIP:/<b>TIP:<\/b>/ig;
    $content =~ s/<a .*?>//ig;
    $content =~ s/<\/a>//ig;
    $content =~ s/<hr>//ig;
    $content =~ s/<head>(.*?)<\/head>//ixsg;
    $content =~ s/INDEX BEGIN.*?INDEX END//isg;
    $content =~ s/SYNOPSIS/Synopsis/g;
    $content =~ s/DESCRIPTION/Description/g;
    $content =~ s/METHODS/Methods/g;
    $content =~ s/  <DT>
                <STRONG>(.*?)<\/STRONG><BR>/
                <dt><span style="font-family:
                Arial;font-style: italic;">$1<\/span>/igx;
    $content =~ s/  <pre>/<pre style="font-family:
                courier,courier new,fixed;">/igx;
    $content =~ s/<\!--  -->//gi;

    # write the new contents
    open my $fh, '>', $file;
    print $fh $content;
    close $fh;
}
```

Using Perldoc to Read the Documentation

If you're on a Unix-like system, you're probably familiar, and possibly even comfortable, with the manpages system, and the man(1) command. Perl comes with a similar tool for accessing its documentation called perldoc. Perldoc functions somewhat similar to manpages, although there are some Perl specific options. Run the perldoc command on itself for more information: perldoc perldoc. Windows© users can use perldoc as well, though they may find the HTML documentation more accessible.

Before accessing the perldoc documentation for WebGUI, you must tell the perldoc command where to find it. Users using the WRE have it easy: sourcing the setenvironment.sh script does all the work. However, if you're not using the WRE, it's not all that difficult: just set the Perl5LIB directory to the top level directory of your WebGUI Perl modules. For example, if you've installed WebGUI in /data/WebGUI, this will be /data/WebGUI/lib.

Installing Third-Party Modules via CPAN

The best part about being a Perl programmer is that you have CPAN©, the largest repository of reusable software ready for you to pick and choose what you need to accomplish your task. That said, how do you get all of those modules ready for WebGUI to use?

The first step is to find the module that you want. The easiest way to do this is to use one of the several web sites set up to search the enormous collection of modules on CPAN (some 13,000 at this writing, undoubtedly more by the time you read this). The two most common sites for browsing what's available on CPAN are http://search.cpan.org and http://kobesearch.cpan.org . Type in a module name, or even a general task description, and take a look at what's available.

Once you find the module that you need, you'll need to install it so that WebGUI knows where to find it. The most common, and simplest, way to do this is using the CPAN command line utility that ships with Perl. WRE users will need to use the CPAN utility that's part of the WRE; non-WRE users will need to ensure that the CPAN command they use installs modules in a place that will function properly.

Session

If WebGUI is a body then the Session object is WebGUI's nervous system: it sends signals throughout WebGUI. The session object is passed into all newly instantiated WebGUI objects, and many non-object-oriented subroutines as well. It can then be used by the object to access the database, interact with the user's request, find out which user is logged in, and more. In this chapter explores how all of the components of WebGUI::Session come together and how they can be used.

User vs Request Session

The most common mistake in dealing with the session object is that developers think it is the user's session. In WebGUI, the session object is a page request session, of which the user is a part. The session is all the subsystems needed to service a page request. It is a connection to a database, an interface to the HTTP protocol, a relationship to the currently logged in user, and more. When dealing with the session object, understand that the user is only a part of it. Really, it exists to service the request.

Managing a Session

Managing a WebGUI session is relatively easy, and once you have a session you can use all of its resources. To start with, you need to open a WebGUI session:

```
my $webguiRoot = “/data/WebGUI”;
my $configFile =  “www.example.com.conf”;
my $sessionId = “aaaaaaaaaaaaaaaaaaaaaa”;
my $session = WebGUI::Session->open($webguiRoot, $configFile, undef, undef,
$sessionId);
```

To create a session, you need to know where WebGUI is installed, and that's the path in the variable called $webguiRoot. You also need a config file name, which is in the $configFile variable in this example. Once WebGUI has this, it will look in $webguiRoot/etc for a file called $configFile. It will read in the contents of that file using WebGUI::Config. The config file tells WebGUI how to act, and how to connect to external resources like a database.

If you pass in a session id, then WebGUI will check the database to see if that session id is already in use. If it is it will attach you to that existing session. Otherwise it will automatically create a new session for you. If you don't pass in a session id, then WebGUI will check with the $session->http object to see if there is a session cookie from which it can read the session id.

Once you have a session you can make use of all the objects contained in a session, which are described in the following sections of this chapter. If you need a bunch of those objects and don't want to reference them by traversing the object tree (which can get quite verbose), then you can use the “quick” method to retrieve them, like this:

```
my ($db, $config, $http) = $session->quick(qw(db config http));
If you need to know the sessionId for any reason you can get it by calling:
my $id = $session->getId;
```

Once you're done with your request, you need to clean up after yourself by calling the following:

```
$session->close;
```

This will shut down the database connection, and clean up all the other session attached objects.

$session->asset

This is a reference to the current asset, if any. The current asset is the one that was requested by the user through the URL. Or, if a non-container asset was just created, then the current asset will be the newly created asset's container. See the WebGUI::Asset API for details.

```
my $id = $session->asset->getId;
```

$session->config

This is a reference to a WebGUI::Config object for this site. See the WebGUI::Config API for details. Config files are stored in /data/WebGUI/etc.

```
my @assets = @{ $session->config->get(“assets”) };
```

$session->datetime

This object provides methods for manipulating WebGUI's Data/Time format epoch (the number of seconds since January 1, 1970). As WebGUI 7.x development proceeds, this will become deprecated in favor of the WebGUI::DateTime object, which uses real dates ("2007-12-31 14:32:11") instead of epochs. However, for at least the rest of 7.x development, this object will remain as is. For details on the methods available in this object see WebGUI::Session::DateTime.

```
my $epoch = $session->datetime->addToDate( $epoch, $years, $months, $days );
```

$session->db

This is a reference to a WebGUI::SQL object. It is created by reading the "dsn", "dbuser", and "dbpass" directives from the $session->config object. Using this object, rather than recreating your connection to the WebGUI database when you need to query the database, performs better.

```
my @usernames = $session->db->buildArray("select username from users");
```

$session->dbSlave

This is a reference to a WebGUI::SQL object. If "dbslave1", "dbslave2", and/or "dbslave3" are specified in the WebGUI config file, then one of them will be randomly instantiated upon call of this method. In this way the load is distributed across all available slaves. However, you can still safely call this method even if there is no slave, because it defaults to a reference to $session->db.

```
my @usernames = $session->dbSlave->buildArray("select username from users");
```

$session->env

While you can always use Perl's %ENV to get at system environment variables, this is a reference to WebGUI::Session::Env, which is just a wrapper for %ENV. WebGUI uses this wrapper because it does some special magic handling of some elements in %ENV depending upon the environment it's in. That way it doesn't have to modify the actual %ENV, which might cause problems. When working with WebGUI code you should never use %ENV. Instead, always use this object.

```
my $remoteAddress = $session->env->get("REMOTE_ADDR");
```

$session->errorHandler

WebGUI currently combines logging and error handling into one object. This may change in the future. For the time being, this is the object you should use. It is a reference to WebGUI::Session::ErrorHandler. The logging aspects of this are run by Log::Log4Perl, and there is a config file called log.conf in /data/WebGUI/etc which sets up the logging rules. There are other methods in this object for setting up debug, doing stack traces, and finally, if you call a fatal() using it, stopping the page request dead in its tracks. Fatal should only ever be called when it would be dangerous to continue. Otherwise you should call error().

```
$session->errorHandler->error(“The user did something that they weren't supposed to.”);
```

$session->form

This is a reference to the WebGUI::Session::Form object, which in turn plugs in to the WebGUI::Form::* modules for input validation. Using this object you can pull form GET and POST variables with or without validation. WebGUI supports multiple levels of validation. Here's how it works. The following is grabbing the data raw with no validation:

```
my $value = $session->form->param(“emailAddy”);
```

This does the same thing, only with basic text field validation (which is almost exactly the same as above):

```
my $value = $session->form->process(“emailAddy”);
```

This is grabbing the value using validation from the proper form control:

```
my $value = $session->form->process(“emailAddy”, “email”);
```

This is grabbing the value using validation and a default:

```
my $value = $session->form->process(“emailAddy”, “email”, “noreply@example.com");
```

And this is grabbing the value using full validation, and a fully formed object:

```
my $value = $session->form->process(“emailAddy”, “email”, undef, \%formDefinition);
```

$session->http

The http object will only work if there was a valid request object passed into the WebGUI::Session->open() call. See $session->request for details on the request object. See the WebGUI::Session API for details on calling open with the request object. The http object is a reference to WebGUI::Session::Http and is the interface for getting and setting HTTP options such as cache headers, mime types, and cookies.

```
$session->http->setCookie(“myCookie”, $value);
```

$session->icon

The icon object is used to generate toolbar icons in a standard way. It takes into account internationalization, accessibility compliance, and other things. See WebGUI::Session::Icon for API details.

```
$toolbar .= $session->icon->cut;
```

$session->id

WebGUI uses a GUID (Global Unique ID) system rather than integers for all of its object and database table keys. It does this to make it easier to move and synchronize data between WebGUI sites. A GUID is a 22 character code consisting of a-z, A-Z, 0-9, and the “-” and “_” characters. It is generated taking into account the site name, the time and date, and a random number. Any time you need a new ID for something you'd use this object (WebGUI::Session::Id) like this:

```
my $id = $session->id->generate;
```

$session->output

WebGUI's output handler processes macros on the output, and also allows you to redirect output for things other than HTTP. For example, you may want to output to a file handle if you're exporting content to static files. It is a reference to a WebGUI::Session::Output object.

```
$session->output->print($html);
```

$session->os

If you ever need to detect what operating system WebGUI is running on you can use the WebGUI::Session::Os object to do so. It's rarely needed as WebGUI's internal subsystems automatically handle the operating system differences for you.

```
my $osName = $session->get("name");
```

$session->privilege

The privilege object provides standard pages and HTTP codes for all common privilege related messages you might need to show a user. For example, if the user is not in the Admins group, but tries to access something that is admin only, you might call the adminOnly privilege message. See the WebGUI::Session::Privilege API for details.

```
my $html = $session->privilege->adminOnly;
```

$session->request

The request object is a reference to an Apache2::Request object. This is created by the WebGUI Apache mod_Perl handler and passed into the WebGUI::Session->open() method at the beginning of the request cycle. Normally, you shouldn't ever need to access this object. It is mainly available to low level subsystems that need to interact directly with the request, like $session->http.

```
my $ifModifiedSince = $session->request->headers_in->{'If-Modified-Since'};
```

$session->scratch

The scratch variable subsystem allows you to attach arbitrary data to the session. It will persist as long as the session exists. See WebGUI::Session::Scratch for API details. The scratch system uses only scalar data, but you can serialize complex data structures into JSON or some other serialized format and store it in the scratch variable.

```
$session->scratch->set("favorite color", "black");
```

$session->server

The server object is a reference to the Apache2::ServerUtil->server object. It is automatically attached to the session by the WebGUI Apache mod_Perl handler. Under normal circumstances it is only used by low level subsystems and should never be needed by you.

```
my $configFileName = $session->server->dir_config("WebguiConfig");
```

$session->setting

The setting object is similar to the config object, except its properties are stored in a settings table in the database rather than in a config file. The disadvantage of this is that it's only able to store scalar config data, or serialized structures (no hierarchy). The advantage of this is that it can be quickly read on every request for settings that might change without a server restart. The config file, on other other hand, is read only at server startup and never again, so if you change something in the config it will require a server restart. For more information see the WebGUI::Session::Setting API.

```
my $value = $session->setting->get(“foo”);
```

$session->stow

The stow object, affectionately called “Stowage” by the WebGUI dev team, is similar to the scratch object except that instead of attaching arbitrary data to the session, it's attaching it to the request. Therefore, as soon as $session->close is called, it goes out of existence. However, because it's stored in memory, it can handle any object or complex reference type without serialization. Despite it working like a global variable, it's a bad idea to use it like that. Globals are always a bad idea. Instead, think of it as a caching mechanism. See WebGUI::Session::Stow for details.

```
my $value = $session->stow->get(“foo”);
```

$session->style

The style object, which is a reference to WebGUI::Session::Style, is used to manipulate the style wrapper for a page. It can be used to apply javascripts and cascading style sheets to the appropriate places in the document, and to load and execute the template used to generate the style.

```
$session->style->setScript(“/path/to/my.js”, { type=>”javascript” });
```

$session->url

The URL object provides a series of utilities for generating WebGUI compliant URL's. It is a reference to WebGUI::Session::Url. It can help you generate extra URL's, make URL's asset compliant, add GET parameters to the end of a URL string, and generate fully qualified site URL's, among other things.

```
my $newUrl = $session->url->append($url, “foo=bar”);
```

$session->user

The user object is a reference to a WebGUI::User object for the user that is currently attached to this session. By default this would be the Visitor user, but other users will be automatically attached if WebGUI detects that they've logged in. You can manipulate this yourself by providing the method with either a user id or a WebGUI::User object:

```
$session->user( { userId => $id } );
$session->user( { user => $user } );
```

You can use the object as a normal WebGUI::User object:

```
my $username = $session->user->username;
```

$session->var

The var object controls the persistence of the session data. It stores things like the session id, the user id of the user attached to the session, the time this session expires, the IP address last used to access this session, and more. In most circumstances you won't ever have to use this object as it's used by lower level subsystems. However, there are two instances where you will use it.

The first is when you want to determine if admin mode is on or not. Or, for that matter, if you want to toggle admin mode. To do that, the methods you would use are these:

```
if ( $session->var->isAdminOn ) {
	$session->var->switchAdminOff;
}
else {
	$session->var->switchAdminOn;
}
```

The second is when you want to explicitly destroy a session. This is different than $session->close, which just disconnects you from a session, but the session itself is still there waiting to be reattached to. You'd want to destroy a session when you're not going to use it again. For example, if you write a command line script the $session->open statement will automatically create a new session for you, but since you're not going to use it again you'll want to explicitly destroy it to clean up after yourself.

To do this, you'd:

```
$session->var->end;
```

Note that even after you end the session, you must still do a $session->close to destroy the objects and close open connections. This is because the var object is only about the persistence of the session, not the live-in memory objects.

Dynamic Loading

The session object is a collection of smaller objects, which are covered in detail in the previous sections. In order to improve performance these objects are not instantiated until they are used. Therefore, the smaller number of these objects you make use of during your request, the better your performance will be. However, once the objects are instantiated, you can safely reuse them over and over again, because they remain cached.

Writing Macros for WebGUI

Perl programmers are lazy people. They like to do only the smallest amount of work they need in order to accomplish a task, and they have a penchant for reusing others' work. The official literature for the Perl programming language even goes so far as to extol laziness as a programmer virtue. The Comprehensive Perl Archive Network©, or CPAN (http://www.cpan.org/) exemplifies this laziness perhaps better than anything else.

WebGUI, being a GUI for web content management and development, also encourages laziness. Among other features, WebGUI provides *macros*, which are special plaintext commands that invoke custom Perl code to perform specific functionality (most often insertion of text / markup). Sounds a lot like an asset, doesn't it? That's because macros and assets share some common characteristics:

- they both comprise specific functionality designed to be reused
- they're both configurable, albeit in different ways
- they're both implemented in Perl and, for the most part, output HTML
- they both have skeleton files from which you can start writing your own custom functionality.

But that's where the similarities end. While your users will be directly viewing assets, they won't be directly viewing macros. That's because macros are included in assets, and are expanded when the asset is rendered.

When invoked, the macro does its thing and (most of the time) returns some content at the point in the asset where it was invoked. This makes it very easy to determine exactly how a macro will affect a page. There's no delayed invocation or anything like that. Just call it and go. But how do you call a macro? Unsurprisingly, it looks rather like a Perl subroutine call. Following are some macro examples:

- ^AdminBar;
- ^FileUrl(/some/path/to/an/image.jpg);
- ^AddSomeNumbers(1, 2, 3);

There are three important parts here: the initial caret (the ^ character), the name of the macro, and the terminating semi-colon.

These three elements comprise macro invocation; without them, nothing will happen. Additionally, for macros that take parameters, the parentheses enclosing the parameters are required. Commas separate parameters. Parameters included, but *not* separated by commas, are taken to be literal strings. For macros that don't take parameters, or for invocations of those that do which don't require passing parameters, the parentheses can be omitted.

How Macros Work

Macros are processed upon asset rendering, at the point in the asset's content where they are found. This simplicity means that it's relatively straightforward to get started developing macros: write some code that produces some kind of markup, and return it. That's it. But of course, there can be more to it than that.

When invoked, the macro's process subroutine (subroutine! It is *not* a method!) is invoked. This subroutine is passed to the WebGUI::Session object, along with any parameters specified for the macro. As pointed out before, invoking a macro works very much like calling a subroutine in Perl. Parameters are all passed as very simple text strings, and thus must be parsed by the process subroutine if they use key=value syntax or something else not directly usable by the process subroutine. Because of how Perl treats string and number data, there doesn't need to be any magical conversion of strings containing numeric values into Perl numbers. The process subroutine's job is to generate some markup and return it to the code invoking the macro. Of course, the macro doesn't always need to return markup, or anything at all. It just won't present any visible content to the user.

The real power in macros comes from the fact that their logic is coded in Perl, as opposed to Javascript. Macros also have access to the WebGUI API, giving them access to fun things like database access and the ability to manipulate anything in WebGUI. Macros also have access to the full power of Perl – all of CPAN is at your fingertips. Caution must be taken, however, as overly expensive operations will delay page loads.

The Macro Skeleton

Like most aspects of WebGUI, there's a skeleton file available for macros from which you can start developing your own custom functionality. Within a WebGUI installation, this file is located at WebGUI/lib/WebGUI/Macro/_macro.skeleton. The process subroutine is called automatically by WebGUI, and must be present for the macro to function.

The job of a macro is to process any given parameters and use them to produce some kind of meaningful output, or to do something useful. This macro skeleton does neither of those, but it does contain everything needed for a fully functioning macro.

Hello World

Let's start with a fully functional, albeit not altogether useful, macro. This macro will output the text “Hello, world!” to page where the macro is called. It's extremely simple and straightforward, but illustrates the main principles involved in developing a macro. There's more to developing a macro than just writing the code, but not much. For this specific example, the “other stuff” is more complicated than the macro itself, but that will only be true for these simplest of cases. This section will cover those details in addition to providing the (short!) code required for the macro itself.

First, the more difficult task: telling WebGUI about your brand new macro. WebGUI looks up information about each macro in its configuration file. Take a look at part of the relevant section from the default configuration file that ships with version 7.5:

```
"macros" : {
        "#" : "Hash_userId",
      "/" : "Slash_gatewayUrl",
      "@" : "At_username",
      "AOIHits" : "AOIHits",
      "AOIRank" : "AOIRank",
      "AdminBar" : "AdminBar",
      "AdminText" : "AdminText",
      "AdminToggle" : "AdminToggle",
        "AdSpace" : "AdSpace",
      "AssetProxy" : "AssetProxy",
      "CanEditText" : "CanEditText",
      "D" : "D_date",
        [...]
     "u" : "u_companyUrl"
        },
```

What you see here is a JSON dictionary. WebGUI uses JSON throughout, and if you're going to be hacking on WebGUI, you'll need to be comfortable with it. Basically, to develop a macro, you only need to know a few things. Macros are specified in this dictionary in key-value pairs. The pairs are separated by colons and enumerated with commas. There must *not* be a comma after the last item in this enumeration of key-value pairs, quite contradictory to Perl, which doesn't care either way. The keys are the strings that will appear in your assets when you invoke the macro. For example, you'd use ^/ or ^@ instead of ^Slash_gatewayUrl. The values, then, are the class names (with respect to the WebGUI::Macro part) of the macros themselves. Thus, if you have a macro whose class was WebGUI::Macro::HelloWorld, and you wanted to invoke it as ^Hello; in your assets, the line would look like this:

```
        “Hello” : “HelloWorld”,
```

Remember to leave off the comma if it's the last key-value pair in the dictionary.

It's very important to change the package name after you've copied the skeleton file. While easy to forget, it will cause your macro to not function.

Now for the code. First, copy the macro skeleton to a new file, such as HelloWorld.pm. Change the package and file names to whatever you specified in the configuration file, and replace the existing process method with this code:

```
sub process {
   my $session = shift;
   return 'Hello, world!';
}
```

The macro receives the session object as a parameter, and returns the message. That's it. The return value from the process sub is what WebGUI displays on the item (asset, snippet, anything capable of containing a macro) where the macro is in the content for that item.

Getting Argumentative: Accepting Parameters

Not all macros can be invoked without any kind of parameter. Some need a bit of extra information to do their jobs correctly. This is possible, and again relatively straightforward. Take a look at an illustrative example. For the purpose of this example, say you want to pass a custom greeting to the user and have it insert the full user name. This could easily be obtained with an appropriate call of the User macro, but will serve this purpose well.

That said, take a look at the code.

```
sub process {
   my $session     = shift;           # get the session
   my $greeting    = shift;           # get the greeting for the user
   my $userId      = $session->user;   # get the user ID
   my $userName    = $user->userName;  # and the user

   # jostle them a bit and inspire them to log in
   if($userName eq 'Visitor') {
      $userName = 'anonymous coward';
   }

   my $markup = <<HTML;
<p>Why, hello there, ${userName}! $greeting</b>
HTML

   return $markup;
}
```

Take a look at this step by step. First is the WebGUI::Session object. Next, is the greeting string. That's a string of text, so no commas should be present in the invocation of the macro, otherwise you'll get part of it in $greeting and the other half will stay in limbo, never to see the light of day. Next, pull the user ID associated with the session, and then get the user's name as stored in his/her profile. Having users log in is a good thing, most of the time, so anonymous users are jostled a bit, and then construct the greeting and return it.

Now for Something Completely Different

The more serious usages of macros have been covered, but you can have a bit of fun with them, too. A perennial favorite is something that will output fortunes to users. Different databases exist for the venerable fortune program, each containing much wisdom. Take a look at the process subroutine for this macro. This subroutine will entail processing key=value argument syntax, rather than the positional syntax used in previous macros. Take a look at the code.

```
sub process {
   # get the session
        my $session     = shift;

   # and the arguments to the fortune command
   my @arguments   = @_;
   my %fortuneArguments;

   # process the arguments passed to the macro
   for my $keyValuePair ( @arguments ) {
      my ($key, $value) = split /=/, $keyValuePair;
      $fortuneArguments{$key} = $value;
   }

   # set up base fortune command (change this if your path is different)
   my $baseCommand = '/opt/local/bin/fortune';
   my @commandArguments;

   if( defined $fortuneArguments{'offensive'} ) {
      push @commandArguments, '-o',
   }
   if( defined $fortuneArguments{'database'} ) {
      push @commandArguments, $fortuneArguments{'database'};
   }

   # finally, run the command
```

```
    my @responseToUser = qx|$baseCommand @commandArguments|;

    # check if something went wrong
    if( ($? >> 8) != 0) {
        return "<p>Sorry, an error occurred retrieving your fortune. Watch out for black cats, 
ladders, and mirrors!</p>"
    }
    else {
        my $output = "<pre>@responseToUser</pre>";
        return $output;
    }
}
```

As always, you start with getting the session object. Next, get the list of parameters. They're to be passed as key=value pairs, separated by spaces, so the macro processing code sees them as “words”. This entails processing the “words”, splitting them up on the '=', and adding them to a hash to keep track of them and their values. Next, configure the base command. The example given here is for a Mac OS X Leopard© installation using Macports©. Most traditional Unix© systems will use /usr/games/fortune or /usr/bin/fortune. Windows© users will probably have to fetch a pre-compiled fortune program to use this example. Following this, the given arguments are checked and command line options to the fortune command are constructed accordingly. Finally, run the command, check for errors, and give either an error message or the fortune to the user. That's it!

Utility Scripts

A utility script is anything outside of the main web interface that uses WebGUI's API. With most other extensible places controlled by WebGUI from the outside, utility scripts control the flow of execution for themselves.

Why Create Utility Scripts?

Utility scripts are primarily useful for maintenance. WebGUI is designed so basically everything can be controlled or changed through the web interface, but there are some tasks that lend themselves to the command line. Sometimes you want to interact with the local filesystem, which isn't safe to expose to the web. You may also want to perform an action across multiple sites hosted on the same server. Other actions are somewhat dangerous and should only be done by someone who knows the risks and can perform backups and be ready to repair damage. Writing your own script is often useful because creating a command line interface is simpler than making a web interface. The same tasks can probably be performed through SQL and shell commands, but by using WebGUI's API your scripts are protected against changes to the underlying structure of the database or file organization. And even when using SQL statements directly, the WebGUI API allows you to connect simply and use various quick methods to simplify your code.

Existing Utility Scripts

Here are some examples of utility scripts and their uses.

See the WebGUI/sbin folder for additional utility scripts.

fileImport.pl

If you have a number of files to add to a server, it can be a slow and inconvenient process to upload them, especially if they are large. If you already have the files on the server, there is an easier way. The fileImport script can load all of the files from a directory (recursively if desired) on the server into the appropriate uploads location. This is much faster and bypasses any limits you might have on uploading. Like most utility scripts, the first parameter needed is the site's WebGUI config file. This lets the script find the database, file locations, and other information about the site. Additionally, it needs the location of the files to import, and the asset to place them under.

```
$ sudo Perl fileImport.pl –configFile=dev.localhost.localdomain.conf \
--pathToFiles=/data/import --parentAssetId=PBasset000000000000002
Starting...OK
End of the childs detection
Building file list.Found file newimage.png.
Found file newpage.html.
File list complete.
Adding files...
    Adding /data/import/newimage.png to the database.
        Create the new asset.
        Setting filesystem privilege. Privileges set.
    Adding /data/import/newpage.html to the database.
        Create the new asset.
        Setting filesystem privilege. Privileges set.
Finished adding.
Committing version tag...Done.
Cleaning up...OK
```

rebuildLineage.pl

While rare, it is possible for the lineage information on assets to get corrupted. Lineage is what allows assets to quickly find their parent or descendant assets, without crawling through every parent to child to child relationship. If something went wrong with this script, it would cause large problems with the site, so it is good to have a backup before doing so.

```
$ Perl rebuildLineage.pl –configFile=dev.localhost.localdomain.conf
Starting...OK
Looking for descendant replationships...
Got the relationships.

Looking for orphans...
No orphans found.

Disabling constraints on lineage...
Rebuilding lineage...
Asset ID            Old Lineage              New Lineage
PBasset000000000000001   000001                   000001
PBasset000000000000002   000001000001                000001000001
...
```

```
lWFxZDyGhQ3-SLZhELa3qw   000001000002000010              000001000002000008
PBasset000000000000003   000001000003                    000001000003
tempspace0000000000000   000001000004                    000001000004

Re-enabling constraints on lineage...

Repairing search index...
Cleaning up...OK

Don't forget to clear your cache.
```

search.pl

This script has two functions. The first is to rebuild the search index. It can also be used to search the index for matching assets. When indexing, it can be told to act on all sites on the server instead of only one. It may be useful to reindex a search if your database search limits are changed. Also, it is possible to, either programmatically or through SQL, modify assets, and this will allow you to create the index so they have to be found through search.

```
$ Perl search.pl --configFile=dev.localhost.localdomain.conf --search Features
4Yfz9hqBqM8OYMGuQK8oLw  Get Features
OhdaFLE7sXOzo_SIP2ZUgA  Welcome

Search took 0.002149 seconds.
```

Useful API Functions

Included here are examples of useful API functions.

WebGUI::Session

This is the core of any utility script. The WebGUI session contains information about configuration files, the database connection, the current user, and more. The first step in any utility script is usually to initialize a WebGUI session.

open method

This initializes a WebGUI session. The needed parameters are the WebGUI root directory, and the name of the config file for the site. There are other optional parameters which are used only when run as a web service.

config method

This gets an instance of the configuration object, and allows you to get or set any option in the configuration file. You may want to add or remove macros from the list. Other options, like the location of the uploads, are also available here.

settings method

This provides access to all of WebGUI's settings, which are normally set in the Settings screen of the Admin Console.

db method

While it is usually safer and easier to use the other API's to interact with WebGUI's database, sometimes you need the power of raw SQL. The $session->db method gives you a WebGUI::SQL object, with easy to use methods like quickArray or quickHash to make database interactions fast and simple.

Most other object creation methods require a session object as their first parameter.

WebGUI::Asset

Here are some method examples for WebGUI::Asset.

new method

If you already know the ID of an asset, you can use the WebGUI::Asset->new method to retrieve an object instance of it, and then use its display or modification methods to perform whatever actions you need. These ID's may have come from an SQL statement, or a getLineage call.

newByUrl method

Similar to new, but given a URL instead of an asset ID.

getRoot method

Sometimes, you don't have any particular starting point, but want to operate on the entire site. All other assets are descendants of the Root asset, so it is a good starting point.

getLineage

getLineage is one of the most powerful functions in WebGUI's API. You can use it to find related assets by defining a set of rules for which to select.

WebGUI::User

Here are some method examples for WebGUI::User.

new, newByUsername, newByEmail methods

These methods get a user object given either a user ID, login name, or email address.

profileField method

This method gets or sets the value of a specific profile field for a user. With a single parameter, the value of that profile field is returned. If a second parameter is given, the profile field is set to that value.

A Simple Utility Script : settings.pl

WebGUI has a good number of global settings, dealing with things like the commit process and user authentication. Usually, these are adjusted through the Setting screen in the Admin Console. It can be useful to control them through the command line or as part of an automated process. While you can connect to the database and use SQL to set these, looking up the needed information and typing out the commands can be time consuming. You're better off creating a small script to do the work for you. While you could start from scratch every time you want to create a script, a utility script skeleton is included with the WebGUI distribution. The first step when creating a utility script is to create a copy of this skeleton, then open it in an editor.

```
$ cd /data/WebGUI/sbin
$ cp _utility.skeleton setting.pl
$ vim setting.pl

use lib "../lib";
use strict;
use Getopt::Long;
use WebGUI::Session;
```

```
my $session = start();
# Do your work here
finish($session);

#-------------------------------------------------
# Your sub here

#-------------------------------------------------
sub start {
   my $configFile;
   $| = 1; #disable output buffering
   GetOptions(
      'configFile=s' => \$configFile,
   );
...
```

There are some subs predefined in the script, but to start with, look at the line near the top saying # Do your work here. At this point the default script has already initialized a WebGUI session, which is what is required to make changes. This example starts simple with the script: make it report the current value of a setting. Below the call to start(), add:

```
my $name = shift @ARGV;
my $value = $session->setting->get($name);
print $value, “\n”;
```

Now you can save the script and run it on the command line, giving it the additional parameter of companyName.

```
$ Perl setting.pl –configFile=dev.localhost.localdomain.conf companyName
My Company
```

In addition to the script name and the setting to retrieve, you need to provide the site's configuration file. Since there can be multiple sites on one server, this is a necessary command for most scripts. The configFile parameter is handled in the start sub, which you can adjust soon. Of the code added, the part that is doing the important work is $session->setting->get($name), which uses the setting object to retrieve the setting.

Just retrieving one setting from the database isn't very useful, though. You can adjust the script to work with multiple settings and set them as well as retrieve. Use a loop for parameters, and allow use = to indicate a new value to store in the database.

```
# Loop through parameters
for my $setting (@ARGV) {
    # Check for an '=' in the parameter
    if ($setting =~ /^(\w+)=(.*)/) {
        # If there was, our regular expression matched the name and requested value
        my ($name, $value) = ($1, $2);
        # Report the old value to help the user detect any mistakes
        print $name . " - old: " . $session->setting->get($name);
        # Set the new value
        $session->setting->set($name, $value);
        # Report the new value as well for easy comparison
        print " new: " . $session->setting->get($name) . "\n";
    }
    else {
        # If there was no '=', we just have a name.
        # Report the value of that setting
        print $setting . " - " . $session->setting->get($setting) . "\n";
    }
}
```

In this case, there are multiple calls on the setting object, both to set and get values. Now run:

```
$ Perl setting.pl –configFile=dev.localhost.localdomain.conf companyName
companyEmail=webmaster@example.com
companyName - My Company
companyEmail - old: info@mycompany.com new: webmaster@example.com
```

Now one setting has been retrieved, and another set in one line.

These setting names aren't always obvious though. You may only know, or be able to guess, part of the name. Unfortunately, the settings API doesn't provide a way to list WebGUI's settings. For this task, you'll have to go directly to the database. The first change you'll need to make is to allow you to request a list of settings. You'll want to add a variable to store this flag:

```
use Getopt::Long;
use WebGUI::Session;
```

```
my $optList;
my $session = start();
```

Next, add the parameter to the command line parser. The start sub uses the standard Perl module Getopt::Long to parse the –configFile parameter. You don't need to get a value like that option; you only need a simple flag. So add an entry to use the --list parameter, and save the flag in your variable:

```
  GetOptions(
      'configFile=s' => \$configFile,
      'list' => \$optList,
  );
```

Now you just need to operate on this flag. The database table 'settings' is what contains your settings, so select all the names that match your parameters. You can adjust your existing setting loop:

```
for my $setting (@ARGV) {
   # If we are in list mode
   if ($optList) {
      # buildArray selects all the results into an array
      my @settings = $session->db->buildArray('SELECT name FROM settings WHERE 
name LIKE ?', ['%' . $setting . '%']);
      # Loop through our list of settings
      for my $name (@settings) {
         # and output their names
         print $name, "\n";
      }
   }
   # Otherwise, operate like before.
   elsif ($setting =~ /^(\w+)=(.*)/) {
```

Now, if you don't specify the list option, it operates exactly like before, but with the list option you can see all the settings related to email:

```
$ Perl setting.pl –configFile=dev.localhost.localdomain.conf --list email
companyEmail
userInvitationsEmailExists
userInvitationsEmailTemplateId
```

```
webguiRecoverPasswordEmail
webguiValidateEmail
```

Package Maintenance: Short but Powerful

WebGUI packages are a convenient way to share content you've created on your site with other WebGUI sites. You may also run a number of sites that share certain assets. You may want to use packages to synchronize these assets, but the process of exporting and importing can be time consuming when done manually. One option is to create a script so you can automate the process. Then it can be a single command to update your sites, or could even be done as a scheduled task. Start again with the utility skeleton.

```
$ cd /data/WebGUI/sbin
$ cp _utility.skeleton package.pl
$ vim package.pl
```

There are several WebGUI modules you'll need to do your work, as well as some regular Perl file modules. There are also three specific pieces of information you'll need from the user:

1. the directory to export to
2. the package to import
3. the location to import to

```
use lib "../lib";
use strict;
use Getopt::Long;
use WebGUI::Session;
use WebGUI::Asset;
use WebGUI::Storage;
use File::Copy qw(copy);
use File::Spec;

my $outputDir = '.';
my $importPackage;
my $importRoot;
my $session = start();
```

Next, you need to set the script up to parse these parameters. In the start sub, you'll need to add the three parameters. Additionally, the process of importing packages will create a version tag. Unless you specifically handle this version tag, it will be left uncommitted. There is sample code provided in the start and finish subs to create and commit a version tag, so uncomment it and choose an appropriate name.

```
sub start {
    my $configFile;
    $| = 1; #disable output buffering
    GetOptions(
        'configFile=s' => \$configFile,
        'outputDir=s' => \$outputDir,
        'import=s' => \$importPackage,
        'root=s' => \$importRoot,
    );
    my $session = WebGUI::Session->open("..",$configFile);
    $session->user({userId=>3});

    my $versionTag = WebGUI::VersionTag->getWorking($session);
    $versionTag->set({name => 'Package Import'});

    return $session;
}

#-------------------------------------------------
sub finish {
    my $session = shift;

    my $versionTag = WebGUI::VersionTag->getWorking($session);
    $versionTag->commit;

    $session->var->end;
    $session->close;
}
```

You now have the appropriate information gathered and can start doing the actual work. After the call to start(), you'll want to choose between two different modes, importing and exporting. If you're importing, you need to create a storage location for your package, add the package file to it, and call the package importing method on your root asset. To export, you'll need to get your asset object from a URL, export it as a package, and copy the package file to the requested location.

```
# First, check if we are importing or exporting
if ($importPackage) {
   # Importing requires a storage location, create a temporary one
   my $storage = WebGUI::Storage->createTemp($session);
   # Add out package file to that storage location
   $storage->addFileFromFilesystem($importPackage);
   # Now we need a base asset to import into.  The user can specify the URL
   # to use as a parent, or we'll default to the import node.
   my $asset = $importRoot ? WebGUI::Asset->newByUrl($session, $importRoot) :
WebGUI::Asset->getImportNode($session);
   # The import process itself is easy, just a single call returning the new asset
   my $newAsset = $asset->importPackage($storage);
   # Report out success
   print "imported $importPackage to " . $newAsset->get('url') . "\n";
}
# Otherwise, we are exporting
else {
   # Calculate full path to output files to first
   my $fullOutputDir = File::Spec->rel2abs($outputDir);
   # Multiple URLs can be specified to export, loop through them
   for my $url (@ARGV) {
      # Find the corresponding asset
      my $asset = WebGUI::Asset->newByUrl($session, $url);
      # If there isn't a matching URL on the site, warn
      # the user and try the next one
      if (!$asset) {
         warn "No asset with URL: $url\n";
         next;
      }
      # Exporting creates a storage location with the package file
      my $storage = $asset->exportPackage;
      # There will only be one file, the package.  Get its filename
      my $file = $storage->getFiles->[0];
      # Calculate the full path we'll be outputing this file to
      my $outputFile = "$fullOutputDir/$file";
      # Copy our package from its storage location to the user specified location
      copy($storage->getPath($file), $outputFile);
      # Report our success
      print "created package $outputDir/$file for $url\n";
   }
}
```

Now exporting is as simple as running the script and giving it a list of URL's.

```
$ Perl package.pl --configFile=dev.localhost.localdomain.conf home
created package ./home.wgpkg for home
```

You can then import this package just as easily.

```
$ Perl package.pl --configFile=localhost.conf --import=home.wgpkg
imported home.wgpkg to home
```

Apache Authentication: When a Script isn't a Script

While writing scripts to run on the command line is a useful way to interact with the WebGUI API, the same techniques can be applied anywhere Perl is used. One powerful place to use Perl, as WebGUI is evidence of, is inside Apache using mod_Perl. With mod_Perl, you have full access to Apache's request cycle and can implement handlers for URL parsing, authentication and access control, content handling, and even logging. WebGUI is implemented as a series of Apache handlers, but takes control of the entire request process.

You may have files available that you want to be handled by Apache's normal request mechanisms, but controlling access to these files would normally mean creating files with user and password information for everyone you want to be able to access them. If you already have users and groups in WebGUI for these people, this is a duplication of effort. With mod_Perl, though, you authenticate against WebGUI's database, leaving the rest of the request to be handled as normal.

This script is a large departure from your previous command line based scripts, so starting with the skeleton wouldn't provide you with anything. In this case, start with a blank file.

$ vim apacheAuth.pl

First, you have to import all the required modules from WebGUI and Apache. You''ll also want to prevent the script from being accidentally run on the command line.

```
BEGIN {                    # Prevent command line execution
   if (!$ENV{MOD_Perl}) {    # This variable is always defined by mod_Perl
      die "This script must be run by Apache 2\n";
   }
}

package Apache2::AuthWebGUI;
use strict;
use warnings;
use WebGUI::Session;         # Load the parts of WebGUI that are needed
use WebGUI::Utility;

use Apache2::Access ();       # Load the Apache modules that are used
use Apache2::RequestUtil ();
use Apache2::Const -compile => qw(OK DECLINED HTTP_UNAUTHORIZED
SERVER_ERROR);
```

Apache has separate phases for authentication and authorization, so your first sub is just to identify the user. Start with the basics of a mod_Perl handler and get the request object and the server object. Next, initialize a WebGUI session. WebGUI sessions know how to handle those objects, so you'll provide them to simplify parts of your code.

```
# Authentication sub
# Add to httpd.conf as PerlAuthenHandler Apache2::AuthWebGUI::authen
# All we are doing is identifying the user
sub authen {
   my $r = shift;                         # Apache Request handle
   my $s = Apache2::ServerUtil->server;    # Server handle

   # Create a WebGUI session, use the same configuration settings as WebGUI
   my $session = WebGUI::Session->open($s->dir_config('WebguiRoot'), $r-
>dir_config('WebguiConfig'), $r, $s);
```

This code handles two types of authentication: basic auth and cookie based. Basic auth is part of the HTTP standard, with the user information being provided in the HTTP headers. The interface for entering this information is determined by the client, though, and cannot be integrated into a website's pages. This makes it less suited for a media rich website, but ideal when working with a command line client, such as wget. Cookie based auth is what WebGUI normally uses. By reusing that cookie, users are allowed to log into WebGUI, then be automatically authenticated to a file store protected by this handler. When using cookie based auth, the session object does all the work for you. It will determine what user is logged in, and you can save that information with Apache.

```
# Two auth modes, by default we allow both to be used
my $mode = $r->dir_config('WebguiAuthMode') || 'both';

# First is cookie mode, uses the session cookie from webgui.
if ($mode eq 'cookie' || $mode eq 'both') {
    # The session will pick out the cookie and find the user it is logged in as
    my $userId = $session->user->userId;
    # If it isn't visitor, they are authenticated
    if ($userId ne '1') {
        # This is a shortcut for the authz handler so we don't have to
        # create a new session or keep track of one across multiple handlers
        checkGroupAccess($session);
        # set the username for the request object
        $r->user($session->user->username);
        $session->close;
        return Apache2::Const::OK;
    }
}
```

For basic auth, you first have to request the authentication information from the client. After doing that, use WebGUI's authentication system to verify the username and password.

```
# Next try basic auth, part of the HTTP standard
if ($mode eq 'basic' || $mode eq 'both') {
    # Check if the browser sent auth info
    my ($status, $password) = $r->get_basic_auth_pw;
    # It didn't or something else went wrong.
    # Tell the browser we need login information.
    if ($status != Apache2::Const::OK) {
        $session->var->end;
        $session->close;
        return $status;
    }

    # The request has a user, find the auth settings for that user
    my $user = $r->user;
```

```
        my $auth = getAuth($session, $user);
        if (!$auth) {
            $session->var->end;
            $session->close;
            return Apache2::Const::SERVER_ERROR;
        }

        # Check the the password, if it checks out, they are logged in
        if ($auth->authenticate($user, $password)) {
            # Our shortcut again for the authz handler
            checkGroupAccess($session, $user);
            $session->var->end;
            $session->close;
            return Apache2::Const::OK;
        }

        # Otherwise, note it for logs
        $r->note_basic_auth_failure;
    }
    # We didn't success, so clean up and return the failure
    $session->var->end;
    $session->close;

    return Apache2::Const::HTTP_UNAUTHORIZED;
}
```

In order to check the identity of a user given its username and password, you need to get a WebGUI authorization object to check it with. The sub getAuth retrieves this for a given user.

```
sub getAuth {
    my $session = shift;
    my $user = shift;
    # need to find out the authentication method, and only use it if valid
    (my $method) = $session->db->quickArray("select authMethod from users where
username=?", [$user]);
    # Valid methods are listed in our config file
    if (!isIn($method, @{$session->config->get("authMethods")})) {
        $method = $session->setting->get("authMethod");
    }
    # make sure the module is loaded
    my $module = "WebGUI::Auth::$method";
    (my $modFile = "$module.pm") =~ s{::}{/}g;
    if (!eval { require $modFile; 1 }) {
        $session->errorHandler->error("Authentication module failed to compile: $module:
$@");
        return;
    }
    # create a new instance of the auth module, or log a failure
    my $auth = eval { $module->new($session, $method) };
    if (!$auth) {
        $session->errorHandler->error("Couldn't instantiate authentication module: $method.
Root cause: $@");
        return;
    }
    return $auth;
}
```

After running this handler, Apache will have verified the identity of the user, and will be able to use that username for any of its normal authorization controls.

Then, configure this authorization handler in your Apache config.

```
# Uses the same configuration as normal WebGUI
PerlSetVar WebguiConfig dev.localhost.localdomain.conf
# Load the file with the code in it
PerlPostConfigRequire /data/WebGUI/sbin/apacheAuth.pl
# Register it to handle authentication
PerlAuthenHandler Apache2::AuthWebGUI::authen

<Location />
    PerlSetVar WebguiAuthMode both  # Allow session cookie or basic auth
    AuthType Basic                  # Setup for basic auth
    AuthName "WebGUI Data Area"
    Require valid-user              # Require a valid login to get access
</Location>
```

This requires that the user can log in in some form, but doesn't check any group memberships. What if you have different users with different access levels, and want to restrict different files to different groups? For that, you need an authorization handler. Authorization takes place after the user is identified, to check if that user should have access to a given resource. For this handler, you need to take the username and check if the user is a member of the groups you are allowing access. When configuring the directory access, you'll specify a whitespace separated list of group ID's that have access.

```
# Check authorization settings to see if user has access
sub checkGroupAccess {
   my $session = shift;
   my $username = shift;
   my $r = $session->request;
   # Session's user if no user specified
   my $user = $username ? WebGUI::User->newByUsername($session, $username) :
$session->user;
   # Read config, split required groups into a list
   my $groupsToAccess = $session->request->dir_config('WebguiAuthRequireGroup');
   return
      unless $groupsToAccess;
   if (!ref $groupsToAccess) {
      $groupsToAccess = [split /\s+/, $groupsToAccess];
   }
   # if the user is in any of the groups, make as success
   for my $groupId (@$groupsToAccess) {
      if ($user->isInGroup($groupId)) {
         $r->pnotes(WebGUIAuthGroupAccess => 1);
         return;
      }
   }
   return;
}
```

This example cheats a bit. This sub is called in the authentication handler, not the authorization handler. This simplifies the session handling. The caveat is that you won't be able to authorize using WebGUI groups without first authenticating using WebGUI, but the odds of wanting that are very low. When this is processed, it saves a simple true or false in the Apache request object. The actual authorization handler, then, only has to check this value.

```
# would be expensive to reinitialize session, so we use the data we saved earlier
sub authz {
   my $r = shift;                        # Apache Request handle
   if ($r->pnotes('WebGUIAuthGroupAccess')) {   # Our saved value
      return Apache2::Const::OK;
```

```
    }
    return Apache2::Const::HTTP_UNAUTHORIZED;
}
```

Configuring this only requires a small change to the Apache config.

```
# Uses the same configuration as normal WebGUI
PerlSetVar WebguiConfig dev.localhost.localdomain.conf
# Load the file with the code in it
PerlPostConfigRequire /data/WebGUI/sbin/apacheAuth.pl
# Register it to handle authentication
PerlAuthenHandler Apache2::AuthWebGUI::authen
# Register it to handle authorization
PerlAuthzHandler Apache2::AuthWebGUI::authz

<Location />
    PerlSetVar WebguiAuthMode both        # Allow session cookie or basic auth
    PerlSetVar WebguiAuthRequireGroup 11  # Only allow access to secondary admins
    AuthType Basic                        # Setup for basic auth
    AuthName "WebGUI Data Area"
    Require valid-user                    # Require a valid login to get access
</Location>
```

You can add additional location sections with their own WebguiAuthRequireGroup directives to have separate restrictions for different paths.

While the main point of WebGUI is to display assets, you can see many places where interacting with the API outside of a web interface is useful. You may want to perform maintenance on your sites, or you might want to use local files on your server. Or, you can use one of WebGUI's subsystems to perform tasks as part of a different program. Any time you need to direct the entire flow of execution, and not have it controlled for you, utility scripts will answer this need.

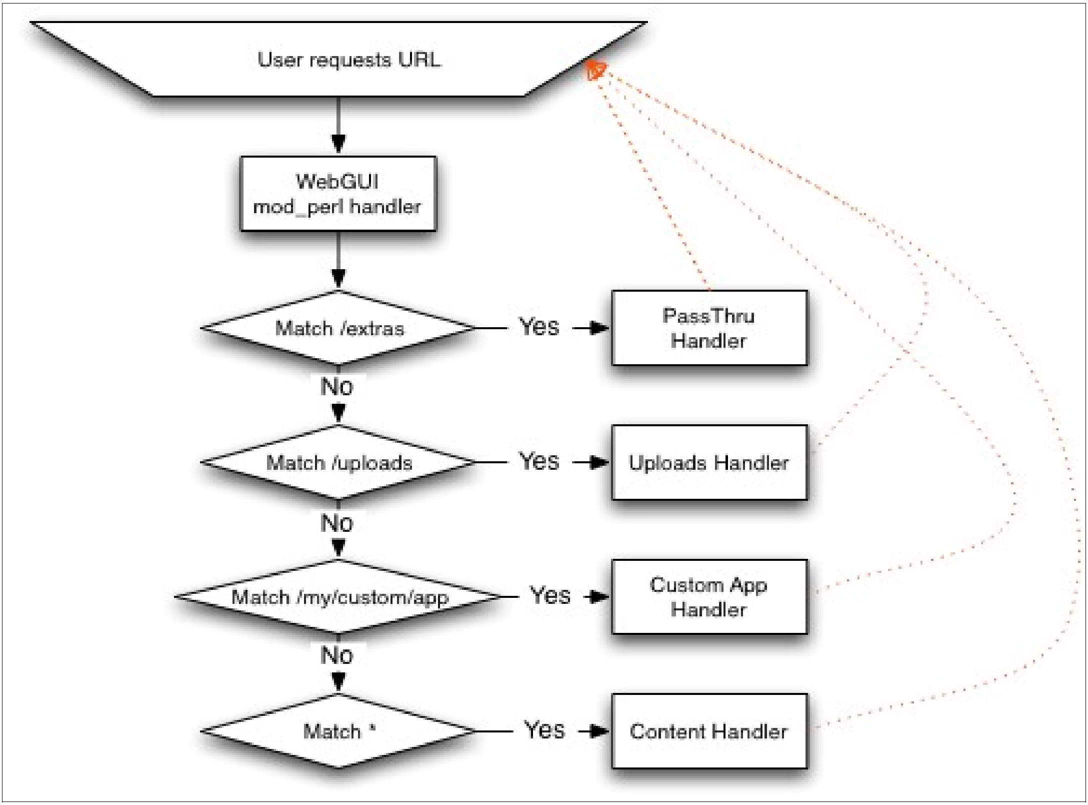
URL Handlers

WebGUI URL handlers serve the same purpose as a mod_perl handler in Apache. They allow you to attach some arbitrary functionality at a specific URL. For example, you can set up some code to apply WebGUI privileges to a filesystem folder full of files, or you can write a handler that will process SOAP requests using WebGUI's data, without going through WebGUI's normal request cycle.

You might think to yourself that since WebGUI is written in mod_perl, is itself a mod_perl handler, and URL handlers are the same as mod_perl handlers, why have them at all? Technically, there's no reason you couldn't write a mod_perl handler in place of a WebGUI URL handler, but as with every part of the WebGUI API, URL handlers set up a mechanism that is a bit more uniform and easy to use for WebGUI tasks. URL handlers give you access to the Apache request cycle, server config, and the WebGUI config, but it hides some of the rudimentary setup that you'd normally have to do so that you can get down to the business of writing your specific functionality. URL handlers also allow you to use Perl-based regular expressions to match URL's rather than the somewhat more obtuse POSIX regular expressions that Apache uses. If you'd prefer to write a mod_perl handler, there's no reason you can't; URL handlers simply give you one additional way to achieve your goals.

The Request Cycle

The following diagram depicts where URL handlers fit into the WebGUI request cycle.



As you can see, the URL handlers are the very first step in the the request cycle.

If your URL handler wants to allow authentication, or other features normally handled by an operation, then you'll need to explicitly include that in your handler, because operations are a content handler, which won't yet be called before your URL handler takes over.

When to Use

There are three main reasons that you'd choose to write a URL handler. You may want to write an application outside of the Asset API. Maybe you don't want to deal with versioning, or you don't wish to conform to the Asset API. Perhaps you want to avoid the asset API because you want to avoid the memory or processing overhead of assets. Or, you may wish to avoid assets because your application is a singleton and there's no reason to ever deploy it to more than one URL.

Another reason to write a URL handler is to integrate an existing application with WebGUI. With a URL handler, you can write the code glue that will bind WebGUI privileges, templates, workflow, or other mechanisms with existing code or applications. This works not only with Perl applications, but also any application with a command line, C, or SOAP API, that you can write a Perl wrapper around.

You may also want to write a URL handler to provide privileges or other processing of external resources, such as a directory of files. Say you have a folder full of PDF's containing financial reports. You only want the people in your accounting department to be able to access those files, and it just so happens that you have a WebGUI group called “Accounting” which includes all the members of your accounting department. Using a URL handler you can place a filter on the folder so that only the accounting department can view those files. If this sounds interesting, stay tuned, because this example will be built later in this chapter.

Configuration

URL handlers are configured by adding a line to your WebGUI config file. The default configuration looks like this:

```
"urlHandlers" : [
   { "^/extras" : "WebGUI::URL::PassThru" },
   { "^/uploads/dictionaries" : "WebGUI::URL::Unauthorized" },
   { "^/uploads" : "WebGUI::URL::Uploads" },
   { "^/\\*give-credit-where-credit-is-due\\*$" : "WebGUI::URL::Credits" },
   { "^/abcdefghijklmnopqrstuvwxyz$" : "WebGUI::URL::Snoop" },
   { ".*" : "WebGUI::URL::Content" }
   ],
```

To add a new URL handler that you've written, simply add a line to the existing configuration. Each line consists of a regular expression pattern to match against, and the class name of the URL handler to use when that pattern is matched. Note that order is important here. If a URL handler that comes before yours has a pattern that matches your URL, then it will handle the URL you want your handler to handle. For the example of the accounting department file folder, the line might look like this:

```
{“^/sales-reports/” : “WebGUI::URL::SalesReports”},
```

After making any config file change, don't forget to restart your mod_perl server.

Examples

The following examples should get you started in writing your own URL handlers.

Hello World

Here's how you'd write a Hello World URL handler.

```
package WebGUI::URL::HelloWorld;

use strict;
use Apache2::Const -compile => qw(OK DECLINED);

sub handler {
    my ($request, $server, $config) = @_;
    $request->push_handlers(PerlAccessHandler => sub {
        my $session = WebGUI::Session->open($server->dir_config('WebguiRoot'),
          $config->getFilename, $request, $server);
        $session->http->sendHeader;
        $session->output->print(“Hello World”);
        return Apache2::Const::OK;
        });
    return Apache2::Const::OK;
}
```

From the Core

WebGUI comes with many URL handlers, so when looking at how to build your own, it may be helpful to have a look at those that have already been created.

For example, WebGUI has a handler called PassThru, which does nothing more than tell Apache that WebGUI doesn't want to handle this URL itself. When Apache sees this, it takes the request back from WebGUI and continues its own processing.

There's another called WebGUI::URL::Content, which itself is just another plug-in mechanism. Instead of handling content based upon any specific URL, it hands off processing to another set of plugins which processes not only the URL, but the entire request to see if they can handle it. Assets are processed via this mechanism. See the chapter on Content Handlers for more information.

WebGUI also comes with a URL handler called WebGUI::URL::Uploads, which determines whether a particular user has permissions to access something in the /uploads folder. If the file has no privileges, or allows for people in the “Visitors” or “Everyone” group to access the file, then the file is served. Otherwise, a permission denied message is sent to the user.

Have a look at the WebGUI::URL::Uploads handler in detail. URL handlers are a standard Perl Module like everything else in WebGUI, so it starts out with the package and use declarations:

```
package WebGUI::URL::Uploads;
use strict;
use Apache2::Const -compile => qw(OK DECLINED NOT_FOUND AUTH_REQUIRED);
use WebGUI::Session;
```

In this example, a number of Apache constants that you want to use are declared. These constants allow you to control how Apache serves the file. You'll see them used as you move further into the handler.

Now, declare a single subroutine called “handler”; the name “handler” is required.

```
sub handler {
   my ($request, $server, $config) = @_;
```

All URL handlers get the Apache $request, and Apache $server objects passed into them, along with a reference to the WebGUI config file for this host.

Next, declare what subroutine should handle privileges:

```
$request->push_handlers(PerlAccessHandler => sub { ... });
```

The reason you do this step, rather than just running the code to determine privileges, is because WebGUI URL handlers are actually called during the PerlInitHandler phase. This allows WebGUI URL handlers to control any aspect of the request that you want. Since the uploads handler is dealing specifically with permissions, it makes sense that it adds a PerlAccessHandler and not a PerlResponseHandler. After all, you want Apache to do what it does best: serve up the files after you determine if the user has the right to access it.

Take a look at what that subroutine looks like:

```
if (-e $request->filename) {
  ...
}
else {
    return Apache2::Const::NOT_FOUND;
}
```

First, check to see that the file requested exists. If not, there's no point in checking privileges on it, so you can return a NOT_FOUND constant. Then, see if the folder containing the file has a .wgaccess file. If not, it's assumed that the file is accessible by everyone, and you can return the constant OK, telling Apache it can send the file.

```
 my $path = $request->filename;
   $path =~ s/^(\/.*\/).*$/$1/;
   if (-e $path.".wgaccess") {
           ...
 }
   return Apache2::Const::OK;
```

Then, read in the contents of the .wgaccess file to see what privileges it says the file should have. If the group to view is 7 (the “Everyone” group) or 1 (the “Visitors” group), then again let Apache serve the file.

```
        my $fileContents;
        open(my $FILE, "<" ,$path.".wgaccess");
        while (my $line = <$FILE>) {
            $fileContents .= $line;
        }
        close($FILE);
        my @privs = split("\n", $fileContents);
        unless ($privs[1] eq "7" || $privs[1] eq "1") {
            ...
      }
```

If the groups are anything else, then you have no choice but to instantiate a WebGUI Session and check to see if the user is a member of the appropriate group.

```
my $session = WebGUI::Session->open($server->dir_config('WebguiRoot'),
$config->getFilename, $request, $server);
my $hasPrivs = ($session->var->get("userId") eq $privs[0]
|| $session->user->isInGroup($privs[1])
|| $session->user->isInGroup($privs[2]));
$session->close();
unless ($hasPrivs) {
    return Apache2::Const::AUTH_REQUIRED;
}
```

If the user doesn't pass the check, you return AUTH_REQUIRED so Apache can display the appropriate error message.

Finally, return the constant OK to Apache. This tells Apache that you are done with the Init phase. If you ultimately decide not to handle the Init phase you could return Apache2::Const::DECLINED instead.

```
return Apache2::Const::OK;
```

Do the least expensive checks first, and go to the most expensive checks only as a last resort. At this level the last thing you want to do is introduce inefficiency. Keep that in mind as you develop your own URL handlers.

Custom Privileges

Let's say your boss tells you that you need to protect a folder full of PDF sales reports so that only people in the accounting and sales departments can view them. Being the enterprising developer that you are, you know that a list of who's in each of these departments is already maintained in your WebGUI intranet, so why reinvent the wheel, just use those lists.

Start out by building a base package:

```
package WebGUI::URL::SalesReports;
use strict;
use Apache2::Const -compile => qw(OK AUTH_REQUIRED);
use WebGUI::Session;
```

```
sub handler {
    my ($request, $server, $config) = @_;
    $request->push_handlers(PerlAccessHandler => sub {
      ... our code will go here ...
        });
    return Apache2::Const::DECLINED;
}

1;
```

Then, flesh out the handler() subroutine with the logic you need. Start with the group ID's for the groups that can access the files:

```
    my $sales = “XXXXXXXXXXXXXXXXXXXXXX”;
    my $accounting = “YYYYYYYYYYYYYYYYYYYYYY”;
```

Instantiate a session:

```
    my $session = WebGUI::Session->open($server->dir_config('WebguiRoot'),
          $config->getFilename, $request, $server);
```

The session already knows who the user is, so you can ask it for the user:

```
    my $user = $session->user;
```

Then add your conditional statement for returning the file based upon user privileges:

```
    if ($user->isInGroup($sales) || $user->isInGroup($accounting)) {
        return Apache2::Const::OK;
    }
    else {
        return Apache2::Const::AUTH_REQUIRED;
    }
```

With just those few lines of code, you're done! See the section above for adding your handler to your config file. Here's the final code:

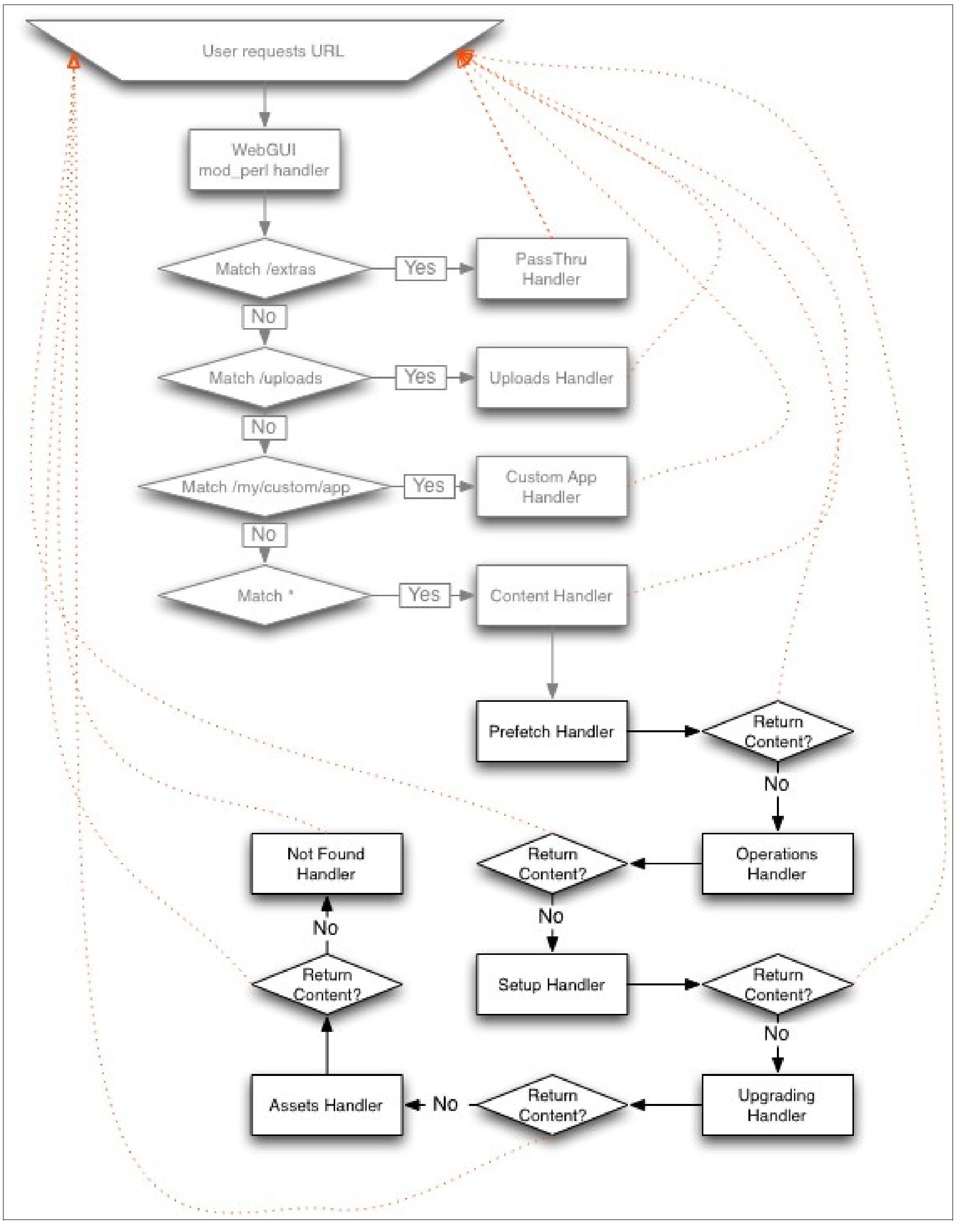
```
package WebGUI::URL::SalesReports;
use strict;
use Apache2::Const -compile => qw(OK AUTH_REQUIRED);
use WebGUI::Session;

sub handler {
   my ($request, $server, $config) = @_;
   $request->push_handlers(PerlAccessHandler => sub {
      my $sales = "XXXXXXXXXXXXXXXXXXXXXX";
      my $accounting = "YYYYYYYYYYYYYYYYYYYYYY";
      my $session = WebGUI::Session->open($server->dir_config('WebguiRoot'),
         $config->getFilename, $request, $server);
      my $user = $session->user;
      if ($user->isInGroup($sales) || $user->isInGroup($accounting)) {
         return Apache2::Const::OK;
      }
      else {
         return Apache2::Const::AUTH_REQUIRED;
      }
   });
   return Apache2::Const::DECLINED;
}

1;
```

Content Handlers

Content handlers give you an easy way to add your own functionality to WebGUI without having to incur the complexity of assets and URL handlers. Content handlers are given a WebGUI session, so you don't even need to set that up. Content handlers, equal to macros, are the easiest plugin you can write for WebGUI.

Request Cycle

Since content handlers are called by a URL handler, they sit in a position directly after the URL handlers. In the diagram below the grey areas are the URL handlers, and the black areas are content handlers.



When to Use

Content handlers should be written whenever you want to write something that doesn't need to directly interact with the request cycle, and doesn't need the power of an asset, but does need to be able to do round-trip requests. For example, a macro is an output mechanism only. You can't post back to a macro;however, with a content handler you can post back to it.

Content handlers need to decide when they will or won't handle a request. They are usually either looking for a specific URL to be requested (like an asset has a specific URL), or they're looking for something in the POST or GET parameters (like operations look for '?op='). If you don't put this sort of condition in, then the handler will always be called, and nothing else will ever get through. See the Hello World example later in this chapter for an example of that.

You may write a content handler as a gateway into some subsystem, such as the Shop content handler acts as a gateway into the WebGUI e-commerce functions. You may also use it to allow a macro to post back to the server, using the macro to display some output, and using the content handler to process the results of the post back.

Configuration

Configuring content handlers is relatively simple. Just add your content handler to the contentHandlers directive in the WebGUI config file. The default configuration looks something like this:

```
"contentHandlers" : [
   "WebGUI::Content::Prefetch",
   "WebGUI::Content::Maintenance",
   "WebGUI::Content::Referral",
   "WebGUI::Content::Operation",
   "WebGUI::Content::Setup",
   "WebGUI::Content::Shop",
   "WebGUI::Content::Asset"
   ]
```

The order in which the content handler is specified is particularly important. Whichever is the first to return a value other than undef is the one that gets to service the request.

To add your own handler, simply add it to the list like this:

```
   “WebGUI::Content::EmailProcessor”,
```

Examples

The following examples should get you started in writing your own content handlers.

Hello World

This is the simplest content handler you can write. Of course, it's also utterly useless. =)

```
package WebGUI::Content::HelloWorld;

use strict;

sub handler {
	return “Hello World”;
}

1;
```

From the Core

WebGUI has many content handlers out of the box. Use these to get an idea of both how to build your own, and what they are capable of. Here's a brief rundown of some of them.

WebGUI::Content::Referral

WebGUI::Content::Referral never outputs anything. Instead, it tracks which site a new visitor to the site came from. As of this writing, this subsystem has very little functionality, and no reporting mechanism, but it does a good job as a proof of concept of how a content handler can be used for simple tracking.

WebGUI::Content::Asset

The WebGUI::Content::Asset handler instantiates the asset subsystem. It matches a URL to an asset using the database, and then hands off the request to the asset to complete processing. This is a great example of a URL based content handler.

WebGUI::Content::Shop

The WebGUI::Content::Shop handler is a good example of a URL query parameter (GET/POST parameter) based content handler. It looks at the query parameters, and if there's one called “shop” then it tries to invoke the commerce system. For example, if a URL looks like /home?shop=cart it will show you the shopping cart.

WebGUI::Content::Setup

The WebGUI::Content::Setup handler looks for a variable called specialState in the settings database table. If it's set to “init” then it will be invoked. This is what drives the WebGUI Site Configuration wizard.

WebGUI::Content::Maintenance

The WebGUI::Content::Maintenance handler also looks in the settings table for the specialState variable, but it's looking for a value of “upgrading”, which is set by sbin/upgrade.pl while it's running. Have a look at how this works. First, initialize the package like any other Perl module:

```
package WebGUI::Content::Maintenance;
use strict;
```

Then, create the required subroutine that all content handlers require:

```
sub handler {
   my $session = shift;
   ...
}
```

Now, check the settings table for your special variable. If it's not what you're looking for, return undef so that the next content handler will process.

```
   if ($session->setting->get("specialState") eq "upgrading") {
      ...
   }
   return undef;
```

If the specialState variable does equal “upgrading,” then you will read in the docs/maintenance.html file and return that back to the user, rather than the requested page, so that the user knows you're conducting maintenance.

```
      $session->http->sendHeader;
      my $output = "";
      open(my $FILE,"<",$session->config->getWebguiRoot."/docs/maintenance.html");
      while (<$FILE>) {
          $output .= $_;
      }
      close($FILE);
      return $output;
```

As you can see from these examples, writing content handlers is quite simple.

Handling a Post from an Arbitrary Form

Let's say your boss tells you that he'd like to collect email addresses from the people that visit the site, but there's a caveat. He wants to be able to display the form on every page, maybe in the style template, and wants the user to be returned back to the page he/she was visiting. You may decide to put the form directly into the style template, or you may use a snippet, or you may even create a macro for it. It really doesn't matter how the form is created. All that matters is that you can use a very simple content handler to capture the email addresses.

Say the form he wants looks like this:

```
<form>
Gimme Yer Email Sucka: <input type=”text” name=”email” />
<input type=”submit” />
</form>
```

You decide that you will look for a form parameter called “storeThisEmailAddress” when it equals “1”, so you add a hidden field to his form, like so:

```
<input type=”hidden” name=”storeThisEmailAddress” value=”1” />
```

Now you write a content handler to deal with this. First, start with a basic package.

```
package WebGUI::Content::EmailProcessor;

use strict;

sub handler {
    my $session = shift;
    ... our code goes here ...
}

1;
```

Then, add a condition that you'll use to activate the handler.

```
    if ($session->form->get('storeThisEmailAddress')) {
        ...
    }
```

If it's true, then you can add your code to store the email address.

```
            $session->db->write("insert into emailStorage (email) values (?)",
                [$session->form->get('email')]);
```

Finally, always return undef from this handler because you want the user to be returned back to the page he/she was previously viewing.

```
    return undef;
```

That's all there is to it! Here's what the final code looks like:

```
package WebGUI::Content::EmailProcessor;

use strict;

sub handler {
    my $session = shift;
    if ($session->form->get('storeThisEmailAddress')) {
            $session->db->write("insert into emailStorage (email) values (?)",
                [$session->form->get('email')]);
    }
    return undef;
}

1;
```

Form Controls

A significant part of traditional thick client application development over the last two decades has focused on so-called GUI's – graphical user interfaces. A GUI is a means of using visual metaphors and event-driven programming to allow a user to visually control and interact with a program. GUI's feature controls (referred to as widgets by some documentation) which are actual interface components with which the user can directly interact. These include buttons, drop-down lists, text input boxes, and things of that sort which have a defined way to visually respond to interface events. Object-oriented control toolkits (such as Microsoft's Windows Forms© framework, or the QtGui module of Trolltech's Qt© framework) typically use object-oriented techniques like inheritance to facilitate understanding and development.

WebGUI, being a GUI for web applications, also supports controls. Just like for thick client toolkits, WebGUI controls must be defined in code before they can be used. However, instead of using bulky designers to lay out the positioning of the controls, code and markup are used to precisely specify where they should be and how they should behave. Once constructed in code, however, the WebGUI UI allows for familiar point-and-click layout.

Form controls are used in WebGUI in a manner similar to how they are used in traditional thick clients: to convey data to WebGUI in a variety of different formats. This can be anything from a standard text entry to a hexadecimal color chooser or even an asset selector. These values are then passed in on form submission and are handled by the appropriate backend code. The backend code is not part of the form control code; the form control code is strictly and specifically concerned with just the interface and the behavior thereof, not with any logic associated with handling the data input into the control. Similar to the rest of WebGUI, and any well-designed application, the interface is separated from the behavior in this way.

In contrast to a WebGUI asset, form controls are for entering data, rather than displaying it. Form controls are generally part of an asset, but only for information to be displayed on that asset or otherwise used. An asset with a form control collects data from the users. After that data has been collected and processed, it's usually displayed on an asset (sometimes the same asset containing the form control, sometimes not). Use a form control in an asset for data collection and an asset for data display.

Before starting development of a new form control, there are several questions to ask. Much of this is common to all development and won't be covered here. However, there is one important question to consider before starting: does the functionality of this form control require any kind of active document technology, like Javascript? Form controls can use Javascript to implement their functionality, and it's important from a design perspective to know whether any potential controls need Javascript before starting development.

Hello World

The simplest example of a form control—or most any computer related task—is the “hello world”. This ensures that the tools (here, form controls) work, and it also serves to familiarize users with the tools and the environment. For form controls, this starts with the _control.skeleton file in the WebGUI/lib/WebGUI/Form directory. This file contains everything a form control needs, and more than it needs for a “hello world” example. The “hello world” form control won't output a form, but rather just the text, “Hello world!” To do this:

- copy the skeleton file to a new .pm file in the Form directory
- change the package line at the top of the file accordingly
- change the definition and toHtml methods accordingly:

```
sub definition {
   my $class = shift;
   my $session = shift;
   my $definition = shift || [];
   push(@{$definition}, {});
      return $class->SUPER::definition($session, $definition);
}
sub toHtml {
   my $self = shift;
   return 'hello world!';
}
```

After these steps, modify an asset's view method and style template to include the control.

```
sub view {
   # other stuff
   $var->{helloWorldControl} = WebGUI::Form::HelloWorld->new($session);
   # other stuff
}
```

Then, include the template variable for the form control in the style template.

```
<tmpl_var helloWorldControl>
```

A Simple Example: Button

In order to better grasp how form controls work, a simple, functional, and easy to understand example works best. Perhaps the simplest of these is the humble, ubiquitous button. The button's entire existence is to serve as a punching bag, taking with stoic acceptance the beatings of hundreds or thousands of users a day. Yet buttons serve a very powerful purpose: without them (and excluding Javascript for the moment), form submissions would not be possible.

All WebGUI form controls live under the WebGUI/lib/WebGUI/Form directory in the WebGUI source tree. Each control is its own separate Perl module; the module for the button is WebGUI/lib/WebGUI/Form/Button.pm. For the sake of brevity (and the assumption that readers of this developer's manual have ready access to a computer with the WebGUI source code), the code for the button control is not reproduced here. However, a few parts of the code do deserve mention.

The first important part is the use base 'WebGUI::Form::Control'; line. All form controls must inherit from WebGUI::Form::Control. This class provides functionality necessary and common to all form controls. There are several methods that a form control should or must override, as well. These include getName, generateIdParameter, and toHtml. The details of overriding these methods will be discussed shortly.

The next important section is the definition subroutine. Like with most parts of WebGUI, this contains meta-information about the class in question, and default values for various attributes. It is, indeed, a definition, a kind of declarative programming. As with all definition subroutines, this one receives several parameters. The first is the class, followed by the WebGUI::Session object, and then the definition that this particular class extends, if any. WebGUI::Form::Button, being a child class of

WebGUI::Form::Control, inherits many fields from WebGUI::Form::Control. However, the button class leaves most of these at their defaults and only defines a few.

Following this is the toHtml subroutine. For most controls, this subroutine does most of the important work. As its name implies, this subroutine generates the HTML required for rendering the control in the browser and any behavior associated with its interface. This can also include Javascript to enhance the behavior of the control. There are several controls which don't define their own toHtml method. Instead, they rely on a parent class to provide the required functionality.

It's time to take a look at what's needed to include a button on a page. Typically, in WebGUI, “page” means “asset”, so this will begin by going over some code and related template markup to include a button on a page. The button will be relatively simple, and will just display a message box saying “Hello, World!” to the user when clicked. In standard WebGUI style, the button will be inserted into the $var hashref passed to the processTemplate method of all assets. The template code will show how to include the button in an HTML document. First, the Perl code:

```
sub view {
	my $self = shift;
	my $session = $self->session;

	# declare $var here, put stuff in it

	$var->{simpleButton} = WebGUI::Form::Button($session, {
		name    => 'simpleButton',
		value   => 'simpleButton',
		extras  => q|onClick="alert('Hello, world!')"|,
		});

	# other stuff this asset needs to do
	# then call processTemplate and pass it $var
}
```

As for the template, it's as simple as designing the surrounding markup and including it in a tmpl_var statement like so:

```
<tmpl_var simpleButton>
```

Now, of course, this particular demonstration doesn't do anything altogether useful. However, it does establish the basic things required to get a button working on an asset. The rest is just additional logic.

For a deeper understanding of how this actually works, take a look at the HTML generated by this example. Again, it's relatively simple, but knowing what it outputs is nonetheless crucial in forming a complete understanding of how the button control functions and interacts with other page elements.

```
<input type="button" name="simpleButton" id="simpleButton_formId"
value="simpleButton" onClick="alert('hello, world!')" />
```

Everything here is rather straightforward: the name and value parameters passed to the constructor are mapped directly to the corresponding values in the HTML. As detailed above, the id parameter is constructed from the name with “_formId” appended. The extras field is appended as a raw string value. This is important to note, because it allows for the specification of custom attributes or behavior for the button that may not be specified in any other way.

As shown above, buttons are actually rather flexible. They serve as starting points for any kind of custom behavior whatsoever. They're similar to their counterparts in thick client applications, but with a few particularities about them resulting from the implementation of HTTP. In stark comparison to thick clients, web applications like WebGUI communicate over HTTP. The important thing to note here is that HTTP is a stateless protocol. What this means is that the web server doesn't keep track of values the user has filled in on various forms between client requests. Thus, other means -like cookies, request parameters, and session information stored in a database – must be used to maintain the information the user filled out on the page between submissions. To this end, any information that needs to persist across page requests must be passed along in one of these ways. While buttons aren't specifically concerned with submitting information, one of button's child classes, WebGUI::Form::Submit, is.

The submit form control is the ubiquitous submit button seen all over the web made into a form control. Its most typical use is to, obviously, submit some kind of data to the server. This requires an HTTP POST, typically, which requires a page load, so the information needs to be submitted along with the POST request, typically as parameters.

Getting Attached: the Attachment Control

The button is relatively simple to understand. To develop a more thorough understanding of form controls, how they interact with pages and how to include them in pages, take a look at a more complicated example. The attachment control, used in WebGUI's powerful and extensive Wiki asset, allows for file attachments to be made to Wiki pages.

The attachment control allows for all the necessary operations on attachments: creating, reading, deleting. These three operations are supported using www_ helper methods. These function in much the same way as www_ helper methods do in assets: WebGUI invokes them based upon parameters passed in on the URL of the request, but with the www_ part stripped. For example, to call the www_delete method of the attachments control, the URL would look something like this:

http://your.company.com/wiki/funWikiPage?op=formHelper;class=Attachments;sub=delete;maxAttachments=5;maxImageSize=100000;thumbnailSize=50;attachments=yfyFDxQam6bEMxbwxu1QEw;assetId=yfyFDxQam6bEMxbwxu1QEw;name=attachments

To fully understand what's going on, analyze this piece by piece. First, the standard protocol prefix, followed by the domain name. Then the location of the wiki asset in the tree, and a page of that wiki ("funWikiPage"). Next is an invocation of a WebGUI operation. The formHelper operation specifies the class name (Attachments, here, expanded to WebGUI::Form::Attachments) and the web-accessible method to call on that attachment (here, delete). Following are parameters required for attachment operation. The attachments parameter, which may be specified more than once, lists the attachments for this particular Wiki page. Next, the assetId parameter specifies the actual attachment to delete.

The quick run-through above details a specific feature of the attachment control. What about how the code works, and how to include an attachment form on a page? Let's start with the attachment definition. The definition contains a few fields particular to the attachment control that are noteworthy. The first of these, maxAttachments, specifies the maximum number of attachments the control will accept at any one time. For the WikiPage asset, this is specified in the Security tab and defaults to zero. The next two fields, maxImageSize and thumbnailSize, describe the maximum and thumbnail image sizes for attachments, respectively.

The next noteworthy aspect of the attachment control is the getValueFromPost method. This method returns an arrayref of asset ID's. When files are uploaded via the attachment control, they are placed in a temporary location. It is the developer's responsibility to move these to a proper location, or else they'll be deleted.

Next is the toHtml method. Just like in button, this method renders the HTML required for this form control to function on the page. Take a look at the HTML generated by this control. There are two parts: an iframe, and the code rendered by the iframe. Here's the iframe code:

```
<iframe
src="/wiki?op=formHelper;class=Attachments;sub=show;name=attachments;maxAttachm
ents=1;maxImageSize=100000;thumbnailSize=50;" style="width: 100%; height:
120px;"></iframe><div id="attachments_formId"></div></td></tr>
```

And here's the HTML generated by the iframe:

```
<html><head> <script src="/extras/AttachmentsControl/AttachmentsControl.js"
type="text/JavaScript"></script>
<link href="/extras/AttachmentsControl/AttachmentsControl.css" rel="stylesheet"
type="text/css" />

     <script type="text/javascript">
      parent.document.getElementById("attachments_formId").innerHTML = '';
     </script>
      </head> <body>
    <div id="uploadForm">
      <a href="#" onclick="WebguiAttachmentUploadForm.hide();"
id="uploadFormCloser">X</a>
      <form action="/wiki" enctype="multipart/form-data" method="post">

      <input type="hidden" name="maxAttachments" value="1" />
      <input type="hidden" name="maxImageSize" value="100000" />
      <input type="hidden" name="thumbnailSize" value="50" />
      <input type="hidden" name="name" value="attachments" />
      <input type="hidden" name="op" value="formHelper" />
      <input type="hidden" name="class" value="Attachments" />
      <input type="hidden" name="sub" value="upload" /> <input type="file"
name="attachment" />
      <input type="submit" value="Upload" /> </form> </div>

      <a id="upload" href="#" onclick="WebguiAttachmentUploadForm.show();">Upload
an attachment.</a>
```

```
            <div id="instructions">Upload attachments here. Copy and paste attachments
into the editor.</div>
        <div id="attachments"> </div> </body> </html>
```

Again, as you can see, the properties of the control are propagated to various parts of the HTML. The actual content of the iframe is a relatively basic file upload form with some hidden fields to maintain state. One thing to note is that the sub parameter of the processed form contains “upload” Again, this corresponds to the www_upload method in the attachments class.

Next is the www_delete web method. As discussed earlier, this method deletes an attachment from the relevant asset. All attachments are stored as assets, so they have access to all of the functionality available to assets: getting ID's, content, exporting, and cleanup methods. The www_delete method uses one of these cleanup methods, purge, to remove the attachment from the asset. When finished, www_delete calls www_show, described next, and returns the value returned from that web subroutine.

The www_show method is rather large, and has a lot going on. As with all but the most trivial subroutines, www_show starts with declaring some variables to be used throughout the sub. If called with a second parameter, it initializes a lexical variable to that parameter (the call in www_delete works this way); otherwise, it takes the parameter list from the URL. Next, after disabling cache control and adding necessary Javascript and style information, the method prepares the markup required for the upload form. Then, the markup used for managing attachments is constructed. After this, the final content, containing the list of existing attachments, the upload form, and management controls, is constructed, and returned.

Last is the www_upload method. Like with www_show, www_upload starts with collecting some information, including information required for saving state. Then there's a call to a very particular method: addFileFromFormPost. This method in the WebGUI::Storage class, obviously, handles uploaded files from form posts. The method saves the content of the file to a location in WebGUI's storage, and returns the filename of where the file was saved. After a security check preventing visitors from doing naughty things with items they've put in tempspace, the method assigns some properties and creates an asset from the uploaded file. After, it adds the new asset's ID to the list of attachments for this asset and calls www_show with the session and this array of asset ID's. Finally, it clears the temporary storage and returns the result of calling www_view, with the session object and the new list of asset ID's.

That's what the methods do. But... how does the blasted thing actually *work*? Let's take a look at it one step at a time, from the browser's perspective. First, before anything else, the toHtml method of the control is called. This generates the iframe. The iframe, as described above, contains a URL that invokes the www_show method. The www_show method is where most of the work takes place. Almost all of the content is generated here, and it serves as the main window, so to speak, of a traditional GUI application. From the layout rendered by the www_show method, all of the useful functionality of the attachment control can be realized. Uploading files, viewing and managing files, deleting files... it's all possible from this view. Each of these functions invokes a different www_ helper method.

Writing Your Own Form Control

So far, an example of a simple control (the button control) and a more complex control (the attachments control) have been introduced, but what are the steps needed to construct your own form control? Like most parts of WebGUI, form controls have a ready-made skeleton file that you can copy and use as a starting point for custom development. This file is located in the WebGUI/lib/WebGUI/Form directory and is named _control.skeleton. Let's examine this file and see how you can use it to put together your very own form control.

First, decide on a name for the control. This will designate the name of the file name, and the package name used to refer to the form in code. For the purpose of this example, call the control SimpleControl. Copy _control.skeleton to SimpleControl.pm. Open up the new file, and take a look at what it provides for you. Each section is covered next.

Before proceeding one step further, change the package name! WebGUI will be unable to find your control and will give you page compilation errors if you forget this most crucial, yet easy to forget, step. It's a very simple thing that, in the excitement of writing new code, developers sometimes completely neglect. Save yourself future headaches (“I'm *positive* the code is correct and it's in the right spot; why can't WebGUI find it? I'm switching to Mambo...”) and change the package name immediately! You'll thank yourself later.

Now that you've saved yourself from embarrassment in front of the boss, take a look at what the skeleton provides and how you can use it. After the Plain Black copyright statement (feel free to add additional legal information to this section pertinent to your organization), the skeleton has various use statements that load various modules and set up the necessary inheritance relationship with WebGUI::Form::Control. Keep the inheritance statement; that's required for the form control to work and it'll do Bad Things® if you take it out.

Next is the obligatory definition subroutine. It's pretty boilerplate, really, but all the parts are there for you to extend in any way you need. Custom fields are easy enough to add, like the thumbnailSize field for the attachments control.

After that is getValueFromPost. This method appeared in the attachments control, but not the button control (there's no information to get from a button; if there's a post at all, you know the button was clicked, and that's the only information the button can provide). Depending upon the functionality of the control you intend to design, you may or may not need this method.

Following getValueFromPost is toHtml. This method, of course, outputs the initial HTML the user sees when they see your control. This could be all they need to see, like with button, or it could be just the beginning, leading to a more complex presentation, like with attachments. Either way, you'll very likely need to change this to output the HTML and/or JavaScript required for your control to function.

That's it. Of course, you'll probably want to divide your functionality into methods and www_ helper functions as designated by your specifications. There are plenty of examples of how to do things already, and it's not uncommon at all for form controls to inherit from one another or use one another to accomplish their tasks. There are dozens of instances of form controls reusing code. Don't be afraid to do the same yourself!

As for using your new control in your control, it's a fairly simple procedure. First, you need to use the package containing the code for your control (remember you had to change the package declaration when you copied the skeleton? You *did* do that, right?), and then instantiate it. It'll look something like this:

```
use WebGUI::Form::SimpleControl;

# somewhere in your asset code
my $formHeader = WebGUI::Form::formHeader(...);
my $simpleControl = WebGUI::Form::SimpleControl->new($session, foo => bar, baz =>
quux);
my $formFooter = WebGUI::Form::formFooter(...);
```

Then, put these objects in your template and you're set.

Authentication Plugins

Authentication in WebGUI allows plugins, providing you the power and flexibility of defining how users authenticate against your site. If you want users coming from a particular IP address to be authenticated automatically, you can do that. If you would like to create a wizard for creating a new account, you can do that as well. Nearly anything is possible with WebGUI's authentication system.

History

WebGUI has allowed authentication plugins since version 5. One of the drawbacks with authentication in version 5 was that you were limited to the set of routines that WebGUI itself expected to call in various places. Thus, you couldn't create or define your own set of rules for authentication. You could only modify the existing rules in an attempt to make them match your requirements. This drawback caused a reevaluation of authentication in WebGUI. WebGUI's developers wanted to provide users with the most flexibility possible while at the same time making authentication plugins easy to develop and manage.

Authentication as it exists in WebGUI today came with the release of WebGUI 6.0. It is object oriented, making it possible to build on existing authentication plugins, which makes them easy and fast to develop. It is also infinitely flexible. You can choose to use or build upon existing methods available through the Authentication package, or you can choose to completely ignore the existing framework and create your own. As you will see in upcoming chapters, the possibilities are limitless.

General Authentication Structure

Authentication is structured such that it is possible to do anything you wish. To make this possible, a single global web operation exists for all authentication methods: op=auth. This method serves as a router into an Authentication plugin. By specifying this at the end of a URL, you tell WebGUI that you wish to make a call within the active authentication plugin. If you wish to see this method, it can be found in WebGUI::Operation::Auth.

In order to call a method from within WebGUI, you must then specify a method within the plugin. You do this by specifying the method after the op=auth call: op=auth;method=login

This calls the login method within the active Authentication plugin. If the current authentication method is the default, or WebGUI authentication, this will call the login method within Auth::WebGUI. Be aware that the method you specify will only be called if it is flagged as "callable" from within the Authentication plugin. More will be covered about this and how to protect methods in your custom authentication plugins later.

If you choose not to enable the methods discussed later in this chapter, some macros may not work, and you may need to change the URL's of some template variables for some functions in WebGUI should you choose to implement them.

WebGUI itself makes only one call within any authentication method. This will be discussed a bit later. For now, you should know that the only method called by WebGUI's core is "init". By overriding the init method, you get to decide what you want WebGUI to do when it calls your authentication plugin. This gives you ultimate freedom in creating an authentication framework within WebGUI.

Writing a Custom Authentication Plugin

Now that you have a basic understanding of how authentication in WebGUI works, it's time to get into the nuts and bolts of writing a custom plugin. The first thing you'll need to decide is what you want your authentication plugin to do and if you can reuse either of the authentication plugins that ship with WebGUI, or any other custom authentication plugin you may have access to. This will determine the starting point of your custom authentication plugin and, as you'll learn later, determines how much of the authentication plugin you actually need to write yourself as some of it could be inherited.

Every authentication plugin in WebGUI is an extension of WebGUI::Auth. If you open this module and look through the various methods, you will see that there are many methods that will be inherited into your custom plugin. You may choose to use, not use, or change the way that these methods work. Most of these methods are discussed in this chapter, so don't spend too much time trying to figure out what is going on here. Once you've decided what you'll be extending (WebGUI::Auth or some subclass thereof), inherit from that class as you would in any regular OO Perl application by using base 'WebGUI::Auth';

Auth Constructor

Like any other object oriented application, you begin writing authentication plugins by creating a constructor. The constructor of your authentication plugin is responsible for setting up a framework for individual authentication sessions. It is important to remember this as authentication deals with one person at a time, not large groups of people.

Also, it is important to understand that you will rarely be responsible for instantiating authentication. WebGUI will typically do this for you once your authentication method is plugged into the system. Typically, you will instantiate authentication in your own custom code, such as if you were building a custom asset or utility script that needed to find out some information about a user's authentication information.

The superclass constructor expects a few things to be passed along in order to instantiate an instance of an authentication plugin:

- WebGUI Session (session): the current active WebGUI session.
- Authentication Identifier (authMethod): the identifier for the authentication method you wish to instantiate. This is covered more later.
- User Id (userId): the GUID (Generic Unique ID) of the user that you wish to instantiate an authentication method for.

You can expect that these items will be passed into your constructor by WebGUI whenever your authentication plugin is instantiated and that the superclass expects them in order to properly set up an authentication framework. Additionally, these items are stored as object data by the superclass which makes them, as well as a few other other convenience instance methods, available to you.

Assume $auth is an instance of a WebGUI Authentication plugin. $auth->authMethod([authMethodIdentifier]) returns or resets the authentication identifier within the current instance of this authentication plugin. This method is important because it determines the authentication plugin that methods will be called from. It is possible to change this value so be careful. Why would you want to change the authentication method? Typically you wouldn't, but in cases where a site runs more than one authentication method, WebGUI will identify the individual's auth method at login time and change the authentication identifier to the proper one.

For example, if a site is configured to use WebGUI's default authentication method, but has some individuals who authenticate through LDAP, when those individuals attempt to log in WebGUI needs to make calls to the proper authentication plugin to properly authenticate them. When they first come to the site, they are essentially logged in as the visitor user, which uses the WebGUI authentication plugin. When the user attempts to log in, WebGUI looks up the authentication identifier associated with the user's account via the username entered, and then changes the authentication identifier currently stored if it is different than what visitor uses. Then, when the init method is called (shown shortly), it is called from the proper module. In the case of users that authenticate via LDAP, that would be the WebGUI::Auth::LDAP module. For those that authenticate via WebGUI, it would be the WebGUI::Auth::WebGUI module.

$auth->session

This returns the session object. You'll need this to retrieve various tasks in your methods.

$auth->user([$user_object])

This returns or resets the user object within the current instance of this authentication plugin. This is an important method for several reasons. Once you've instantiated your auth method, you should never have to create the user object again, which is very slow. Additionally, you use this method to reset the user that is currently associated with this authentication session. Why would you want to do that? Take the following example:

> Someone comes to a WebGUI site using WebGUI's default authentication plugin and does not have a valid WebGUI session cookie identifying them. The user is logged in as the “Visitor” user (userId 1) by default. Once the user successfully logs onto the site (however that might happen), you want to change the userId that is associated with the current authentication framework to that of the user that just logged in. To do this, after a successful authentication call: $auth->user(WebGUI::User->new($auth->session,$userId)).

You can see an example of this in WebGUI::Auth::authenticate.

$auth->userId

This is a convenience method which returns the userId for the user currently set in the authentication method. This is equivalent to $auth->user->userId.

$auth->username

This is a convenience method which returns the username for the user currently set in the authentication method. This is equivalent to $auth->user->username.

The constructor also initializes some object data that can be used to retrieve or set warning and errors from the superclass. Many times there is a need to return varying amounts of data to a method call. In authentication this often happens in the form of returning success, or an error and an error message. Rather than having to do things like return zero on success or a message if an error happened, or true and undef in the case of no error, the authentication system allows you to simply return true or false and set object data containing any errors or warnings that you might want to throw.

$auth->error([$errorMessage])

This returns or sets the current error message stored in object data.

$auth->warning([$warningMessage])

This returns or sets the current warning message stored in object data.

Let's say you are validating that a password entered complies with some custom rules you require for entering passwords. There could be many reasons the password doesn't pass the test. Instead of having to name your method something like passwordIsNotValid and return false if the password is valid, you can simply return true or false and set a warning or error in object data that can be used if necessary:

```
my $false = 0;
	if(length($password) < 10) {
		$self->error(“Passwords must be 10 characters long”);
		return $false;
	}
```

Finally, the constructor is where you need to declare the methods in your authentication plugin are "callable" via a URL. You do this using the following API method.

$auth->setCallable(\@callableMethods)

This security feature enables you to write an unlimited amount of methods in your authentication plugin, but have a limited number that can be called in the form of op=auth;method=yourMethod. Methods that are not declared in this fashion will not be allowed to be called from the URL.

Once you have created an instance of your authentication method, use the setCallable method to determine the additional methods you would like to be called from the URL.

```
        my $auth        =  $class->SUPER::new(@_);
        my @callable =  ('login','logout');
        $auth->setCallable(\@callable);
```

This enables the login and logout subroutines of your plugin to be called via URL. Additionally, you can determine if a method is in the callable stack by using the following API method.

$auth->isCallable([methodName])

Take a look at the typical constructor of a custom authentication method.

```
        sub new {
                #Shift the class off the argument stack
                my $class = shift;
                #Call the superclass method passing in the remaining argument stack
                        my $auth = $class->SUPER::new(@_);
                #create a callable array with the methods we'd like to be url accessible
                my @callable = (
                        "login",
                        "logout",
                        "createAccount",
                        "createAccountSave",
                        "deactivateAccount",
                        "deactivateAccountConfirm",
                        "displayAccount",
                        "displayLogin"
```

```
		);
			#Tell the auth method that the methods in the callable array are url
accessible
		$auth->setCallable(\@callable);
		#Return the object
		return $auth;
	}
```

You'll notice that the first thing done is call the superclass method which returns a valid authentication object of your type. Then, set the callable methods that you will create or use via inheritance and return a reference to the blessed authentication plugin. You can now continue to build the plugin, making customizations as necessary, as you will see in the following sections.

init()

As mentioned earlier in the chapter, the init method is the only one that gets called by WebGUI. This method is the entry point for any authentication plugin. It is here that you will determine what happens when a user tries to log in or attempts to access a privileged page as the visitor.

By default, the init method of the Auth superclass calls its displayLogin method. You can choose to use this method as is, override this method in your own plugin and make it work the way you need it to work, or you can override init completely in your plugin and make it do something completely different.

Let's say you want people coming from a particular IP address to automatically have access to a portion of your site that normal visitors should not. If someone tries to access the portion of the site from another IP address, you want to present the normal login page.

Every asset has three global privilege properties: owner, groupToView, and groupToEdit. If the user attempting to view the page is not the owner, and does not belong to either of the groupToView or groupToEdit groups, WebGUI will call the init method to determine what to do next.

In this example, you would want to create another visitor-like user, that you'll call “ipvisitor”, using WebGUI's user management system, and add that user to the groupToView group you've defined for the particular portion of the site you'd like to restrict to those users.

Now, override the init method in your custom auth plugin and check the IP address of the user attempting to access the site. If the IP address matches the one you check for, you will automatically log the user in as the ipvisitor. If not, you'll return the displayLogin page (this method is discussed in the next section).

```
	sub init  {
		#shift the object off the argument stack
		my $self           = shift;
		#Create a session variable
		my $session     = $self->session;
		#Retrieve the allowed IP address from the authentication settings
		my $allowedIP = $self->getSetting("allowedIPAddress");
		#Get the IP address of the visitor
		my $visitorIP = $session->env->getIp();

			#Log the user in of the visitorIP address is the same as the  allowed IP
address
			#and the user is currently the visitor
		if($self->userId eq “1” && $visitorIP eq  $allowedIP) {
			#Retrieve the userId we wish to log users in as from the
			#authentication settings
			my $ipvisitorId = $self->getSetting("ipvisitorId");

			#If the ipvisitorId is not set in the settings, fall through
			if ($ipvisitorId) {
					#Create the ipvisitor user object
				my $ipvisitor = WebGUI::User->new($session,$ipvisitorId);

				#Set the ipvisitor as the user in the auth method
				$self->user($ipvisitor);
				#Log the user in
				return $self->SUPER::login;
			}
		}

		#Otherwise return the displayLogin method
		return $self->displayLogin;
	}
```

Start as you would start building any instance method in an object by shifting off the object from the argument stack. Then, use the getSetting method of the superclass to return the IP address you've stored in the authentication settings (how to do this is later in the chapter). Use the getIp() method from the WebGUI::Session::Env API to retrieve the IP address of the current user.

Then, check to see if the IP address of the user matches the IP address stored in the authentication settings. If they do, you'll get the userId of the ipvisitor user you mentioned earlier, which you also have stored in the authentication settings. If the userId has been set (you don't want to try to operate on a userId that doesn't exist), you'll create an instance of the ipvisitor user object and set that as the new user object for this authentication session. Once that is done, you can simply return the login method of the superclass which will log in the user and forward him on to the page that was trying to be accessed (again, this method is discussed later in the chapter).

If the IP addresses don't match, simply stick with the default logic and forward the user on to the displayLogin method of the authentication plugin.

As you have probably figured out by now, the possibilities in this method are limitless. As the single point of entry into WebGUI's authentication system, the init method gives you the power to make your authentication plugins extremely flexible.

displayLogin()

If you don't choose to override the init method, the displayLogin method will be returned. This method is really a convenience method which returns a template containing the core form elements that are used in almost every traditional login page. Typically, you will not want to override this method, but instead you will want to extend it with some options that are available from the superclass.

$auth->displayLogin([postMethod, templateVariables])

“postMethod” is a string which indicates which callable method should be posted to when the user makes a request. By default, postMethod is set to “login”, a method you have seen in the previous section and one that will be covered in the next. If you choose to pass in your own postMethod, the form will be generated so that when the user fills in his username, password, and whatever else you might specify, it will post to whichever method you specify. Keep in mind that this method must be listed in your callable methods. For example, $auth->displayLogin('doLogin'). This will attempt to post to the doLogin method of your custom authentication plugin.

“templateVariables” is a hash reference containing template variables that you can create in your extension of displayLogin that will be exported to the template. By default, the following template variables are available from the displayLogin superclass method:

Variable	Description
title	Internationalized title of the login page.
login.form.header	<form> open tag containing action and method.
login.form.hidden	Hidden fields which specify which page to post to.
login.form.username	Text field for accepting username input.
login.form.username.label	Internationalized label for username.
login.form.password	Password field for accepting password input.
login.form.password.label	Internationalized label for password.
login.form.submit	Submit button for posting login information.
login.form.footer	</form> closing tag.
anonymousRegistration.isAllowed	Boolean indicating whether or not users can anonymously register for the site (this is a global authentication setting).
createAccount.url	A URL which links to the createAccount authentication method.
createAccount.label	Internationalized label for the create account href.

You extend these by passing in a hash reference of your own custom template variables that will be used on your display login page. For instance, say you wanted to add password recovery to your authentication plugin. You would want to display a URL linking to your password recovery method, and possibly a label. In order to get those items into your template, you need to pass them along as a hash reference to the displayLogin superclass method. It is important to note that you cannot change the template variables in the superclass. If you ever want to change something that was in the superclass variables, you need to create entirely new template variables and use those instead. You could also choose to override the displayLogin method completely and make it do whatever you need it to do.

Let's extend the displayLogin method and add a few things. First, if the user is already logged in, you'd rather not display this page. Instead, bounce the user to the displayAccount page, which will be discussed a bit later. Next, add an error message that you can display to the user in the case that login fails for some reason. Finally, add your own password recovery method. So, you'll add some extra template variables which link to the password recovery method, which is written later in the chapter.

```
sub displayLogin {
    #shift the object off the argument stack
    my $self       = shift;
    #shift off any message that we might have passed in from a failed login
    my $message = shift;
    #Create a session variable
    my $session   = $self->session;
    #Create an empty has to use for our template variables
    my $vars       = {};
    #Return the display account page if the user is already logged in
    if($self->userId ne "1" && $self->userId ne $self->getSetting("ipvisitorId")) {
        return $self->displayAccount($message);
    }
    #Add some template variables to the empty hash we created
    $vars->{'login.message'} = $message;
    $vars->{'recoverPassword.url'}
                    = $session->url->page('op=auth;method=recoverPassword');
    $vars->{'recoverPassword.label'}
                    = "Forgot your password?  Click here";
    #Return the superclass displayLogin method with our additional template
    #variables
    return $self->SUPER::displayLogin(undef,$vars);
}
```

Start out by shifting the session object off the argument stack as usually done. Also, shift a message off. If an error occurs in the login post method, you'll export the message to the template to display to the user. Now, check the userId for the authentication session. If the userId is not “1”, which is the userId of the visitor, and the userId is not that of the ipvisitor (ipvisitors should be able to log in as another user account), that means that the user is already logged in. You don't want users who are already logged in to log in again, so show them the displayAccount form instead and pass it any message that might be forwarded. If this is the visitor or ipvisitor user, then you'll set the login message template variable along with a URL and label for your custom password recovery method and then return template returned by the displayLogin method of the superclass.

One final thing to note for the displayLogin method is that it is different from a lot of the other authentication methods in that it is usually called before you know which authentication method to use. In order for the displayLogin method of your authentication plugin to be called, you must be sure to set the default authentication method for the WebGUI site to your authentication plugin. You do this on the authentication tab of the Settings screen within WebGUI's Admin Console. For more information on administering WebGUI, see the *WebGUI Administrators Guide*.

getLoginTemplateId()

With all this talk of template variables and templates returned by the displayLogin method, you are probably wondering how to set the template you would like your method to use. If you have overridden the displayLogin method completely, this isn't a problem, but if you have extended the superclass method, in the previous section, you need a way to do this as templateId wasn't something that could be passed into the superclass method.

You do this by adding a getLoginTemplateId method to your custom authentication asset which simply returns the ID of the template to use. This is typically a setting with a default templateId set to the templateId of the default template which you provide with your authentication method:

```
sub getLoginTemplateId {
	#Shift the class off the argument stack
	my $self	= shift;
	#Retrieve the templateId from the authentication settings
	my $template = $self->getSetting("loginTemplateId");
	#Return the template stored in the settings or the default template
	return $template || "IPtmpl0000000000000003";
}
```

You create similar “get” methods for templates from superclass methods listed in the table below. You should also create similar template methods for templates from your custom methods making them easier to subclass.

Superclass Method	**Template Method**
displayAccount	getAccountTemplateId
createAccount	getCreateAccountTemplateId

login()

Unless you have specified otherwise, the login() method is called when a user submits the displayLogin form. This method performs the actual login for the user, so it is almost always the case that you need to extend or override this method and add the actual authentication logic.

The superclass login method is very general in order to make it easy to authenticate users however you like. This method changes the user that is currently in WebGUI session to the user currently stored in the authentication object instance. If you look inside the login() method of the WebGUI::Auth, you will see a line of code that looks like this:

```
$self->session->user({user=>$u});
```

This one line of code essentially performs the login. This is important to know if you plan to completely override the login method and make yours do something else. You will always need this code in some form or other in your login routine.

In addition to performing the actual login, the login() method also gives the user karma if the global karma setting is enabled. It also logs the login in the userLoginLog database table. This table stores all current login sessions which are displayed in the WebGUI's Login History administrative interface.

The default login() method also handles changing the encryption after login if you are using an SSL certificate, redirects the user to the page he/she was trying to access when he/she was prompted to log in, and runs any command line script you may have configured to run when a user logs in.

Since WebGUI has global authentication settings, you should handle each of the above when logging a user in. You can choose to do this yourself if you override the login method (or choose to bypass it completely), or you can simply extend the current login method and add your own logic.

As stated earlier, you will almost always extend or override this method as it doesn't actually authenticate the user. It is your responsibility to decide how users are authenticated. Many different types of authentication exist. In WebGUI, you authenticate by matching a username and password MD5 hash combination against what is stored for the user in the authentication database table (this is discussed later with creating accounts). The LDAP authentication method looks up the distinguished name for the user in the authentication database table and attempts to bind to the LDAP server with that distinguished name.

A convenience method exists in the superclass for very basic authentication. You may choose to use it or extend it as it does some standard checks against the user table that you would probably do anyway.

$auth->authenticate($username);

Every user in WebGUI, regardless of the authentication method you are using, must have an entry in WebGUI's users table. The authenticate method in the superclass looks up the user by username. If the user does not exist, an error is set in the object's error data and false is returned. If the user does exist, but is not active, an error is set in the object's error data, the login attempt is logged, and false is returned. Finally, if the user does exist and is active, the user object is created, set for the authentication session, and true is returned.

Say you wanted to authenticate users via a database; however, you don't feel the MD5 hashing algorithm is secure enough for your passwords. Instead, you'd like your passwords hashed by something with a longer key like SHA512. There are actually two steps to this. The first is the actual authentication. Later in this chapter, saving the password hash will be discussed along with creating accounts.

To write the SHA512 password authentication, start by using the Digest::SHA module that comes with Perl in your authentication plugin. Then, extend the authenticate method to accept a password in clear text. You can then SHA512 encrypt the clear text password and compare that to what you have stored in the database. If they match (and you should also check to make sure that the password entered is not nothing), you will allow the authentication to continue. If they don't match, you'll set an error message in the object data and return false to the login method.

```
	sub authenticate {
		#shift the object off the argument stack
		my $self	= shift;
		#shift the username off the argument stack
		my $username = shift;
		#shift the password off the argument stack
		my $password = shift;
		#validate the username using the superclass method
		unless ($self->SUPER::authenticate($username)) {
			#The superclass method will log the error for us, return false
			return 0;
		}
		#SHA-512 hash the password
		my $sha512pw = Digest::SHA::sha512_base64($password);
		#Get a hash reference of all the authentication data for this user
		my $userData = $self->getParams;
		#if the password is empty or the current identifier in the data does
		#not match the hashed clear text password, the password is not correct.
		if ($password eq "" || $sha512pw ne $userData->{identifier}) {
			#Return an error to the user
			$self->error("Username/Password combination is not correct");
			#Reset the user object to the visitor since login was unsuccessful
			$self->user(WebGUI::User->new($self->session,1));
			#Return false
			return 0;
		}
		#Username and password combination passed, return true.
		return 1;
	}
```

As usual, start by shifting all of the arguments off the argument stack. In this case, you are accepting two arguments: username and password. You then validate the username by calling the superclass authenticate method, which was discussed earlier. If this fails, simply return false to the login method as any errors will already be in object data for you to return. If user authentication passes, you then want to validate that the passwords match. Start by SHA-512 hashing the clear text password that was passed in by the login method and contains the value that was posted in the password field. Then, use another superclass API method to retrieve all of the authentication data for the current user.

$auth->getParams();

This method returns a hash reference containing all of the data from the authentication table for the current user and authentication plugin. Note that by validating the username first, you are assured that the current user is that of the username entered in the form field as it gets reset in the authenticate method of the superclass. When you call getParams() in your extension of authenticate, you receive the data from the user attempting to log in rather than that of the visitor user.

From this hash you retrieve the identifier field from the database which stores your hashed password. You can then compare that to the hash of the clear text password that was passed in. If the password is empty or the hashes don't match, you know that the password entered is incorrect. Then, set an error message in the object and reset the user in the authentication session to the visitor object. If you don't do this step, the user will be logged in even though he wasn't properly authenticated against the system. You can then return false to the login method which will relay your error message to the user, as you will see shortly. If the passwords do match, simply accept the login performed by the superclass method and return true to the login method.

Now, simply call your authenticate method, pass it the username and password that were submitted from the displayLogin form, and decide what to do on a successful and unsuccessful login. In the example, if authentication is successful, you'll simply let the superclass do the work of logging the action and getting the user to the right page. If authentication is unsuccessful, you'll return the user to the displayLogin page and output an error message.

```
	sub login {
		#Shift the class off the argument stack
		my $self      = shift;
		#Create a session variable
		my $session  = $self->session;
		 #Get the username from the form post
		my $username = $session->form->process("username");
		 #Get the password from the form post
		my $password = $session->form->process("identifier");
		#If the username and password combination don't authenticate
		unless($self->authenticate($username,$password)) {
			#Set the http status of the page returned
			$session->http->setStatus("401","Incorrect Credentials");
			#Log a security warning to the error log
			$session->errorHandler->security(
				"login to account $username with invalid information."
			);
			#Return the displayLogin page and pass it the error for display
			return $self->displayLogin($self->error);
		}
		#Authentication was successful, continue logging in the user
		return $self->SUPER::login();
	}
```

logout()

Now that your users can log in, you should probably give them a way to log out. Fortunately, this process is so simple it almost never requires extending the functionality provided by the superclass. In most cases, you don't need to provide a logout method.

In cases where you do need to add or change the default logout functionality, it is a good idea to extend the superclass logout method rather than override it completely as it handles ending the current user's session, setting the current user to the visitor user, and running any logout script that might be necessary.

Suppose that you wish to redirect the user to the home page whenever she/he logs out of the website. To do this, simply extend the functionality of the logout() method and set the http header to redirect the user to the site URL after the logout is complete.

```
	sub logout {
		#Shift the object off the argument stack
		my $self      = shift;
		#Create a session variable
		my $session  = $self->session;
		#Log the user out with the superclass method
		$self->SUPER::logout();
		#Redirect the user to the homepage of the website
		my $homepage = $session->url->getSiteURL;
		$session->http->setRedirect($homepage);
		#Return the default behavior
```

```
                return;
        }
```

If you do not make the logout method callable, the default templates for the Login Box and Login Toggle macros, as well as the account options list, will contain links that will not work properly.

Authentication Data

Now that your users are able to log in and log out of your site, it's time to allow user accounts to be created, updated, and deactivated. Before getting into that, it's important to understand how data related to users, and authentication in general, is stored in WebGUI.

Settings

Let's start by talking about local and global authentication settings. You've already seen examples of retrieving local authentication settings in this chapter using the getSetting() method: $auth->getSetting([$settingName]);

All settings in WebGUI are stored in the settings table which looks like this:

Field	**Type**	**Null**	**Key**
name	varchar(255)	No	PRIMARY
value	text	Yes	

Notice that there is no concept of namespace, which creates a small problem with authentication in that each plugin can have its own local settings, but it's not possible to reuse the same setting name as it serves as the primary key of the table. To overcome this, the getSetting method of the superclass automatically namespaces your local settings for you by placing the name of your authentication identifier (which must be unique to each WebGUI site) in front of the name of the setting passed in, capitalizing the first letter of the setting name so that it becomes camel cased. Thus, $auth->getSetting("ipvisitorId"); for the Yourplugin authentication plugin returns the setting yourpluginIpvisitorId.

Global authentication settings are returned by calling the get method from the settings object within the session object.

```
my $globalSetting = $self->session->setting->get(“encryptLogin”)
```

Global and local authentication settings are found in the Settings screen of WebGUI's administrative interface. Global authentication settings are found on the User tab. Here you'll find settings such as whether or not to allow Anonymous Registration, what workflow activity to run when a new regular or admin user is created or updated, whether or not karma is enabled and how much karma should be given for each login, and so on. WebGUI's Authentication superclass handles the display and processing of all global authentication settings, so if you completely override something, it will be up to you to determine what to do with these settings. You may choose not to use them at all, but they will not be removed from the user interface, making it is easy to create something that is confusing to use. For instance, if you decide to override the login method and completely ignore karma, the enable karma setting will not work for your plugin regardless of whether or not it is turned on. You cannot add global authentication settings without directly modifying the core of WebGUI.

Local authentication settings are found on the Authentication tab of the Settings screen and are located within the box titled with the name of the authentication identifier for each plugin. These settings are specific to each authentication plugin, so anything custom you wish to add is done here. Recall that previously in the chapter the getSetting method was used to retrieve the ipvisitorId, allowedIpAddress, and all of the templateIds.

editUserSettingsForm()

To add local settings for your authentication plugin you must override the editUserSettingsForm() method in the superclass. If you look, this is an empty method because, by default, WebGUI assumes your authentication method has no settings. Another important thing to note is that this method does not need to be callable, as it will never be called directly. Instead, this method is executed for each of the authentication plugins installed on your site (to see the executing code, look at WebGUI::Operation::Settings::www_editSettings()). If you do not override this method, your auth method will appear on the Authentication tab of the Settings screen as an empty box.

To override this method, you must create a WebGUI::HTMLForm object and return the printRowsOnly method as the calling function to write the raw form elements directly to the page. In this example, you'll want to create settings for the ipvisitorId, allowedIpAddress, and the various templateIds of the authentication plugin. It is vital that you namespace your settings as you do not want your form elements to share the same name as other form elements on the Settings screen. Use the namespace rules described above to ensure that you will not have naming conflicts, and that your settings can be retrieved with the getSetting() method mentioned earlier.

```
    sub editUserSettingsForm {
        #shift the object off the argument stack
        my $self = shift;
        #Create a session variable
        my $session = $self->session;
        #Create the WebGUI::HTMLForm object
        my $f = WebGUI::HTMLForm->new($session);
        #Create the form element for the ipvisitorId setting
         $f->user(
            name  => "ipIpvisitorId",
            value  => $self->getSetting("ipvisitorId"),
            label => "IP Visitor User",
        );
        #Create the form element for the allowedIPAddress setting
        $f->text(
            name  => "ipAllowedIPAddress",
            value  => $self->getSetting("allowedIPAddress"),
            label   => "Allowed IP Address",
        );
        #Create the form element for the accountTemplate setting
        $f->template(
            name         => "ipAccountTemplateId",
            value        => $self->getSetting("accountTemplateId"),
            namespace => "Auth/IP/Account",
            label        => "Display Account Template",
        );
        #Create the form element for the createAccountTemplate setting
        $f->template(
            name         => "ipCreateAccountTemplateId",
            value        => $self->getSetting("createAccountTemplateId"),
            namespace => "Auth/IP/Create",
            label        => "Registration Template"
        );
        #Create the form element for the loginTemplate setting
```

```
		$f->template(
			name		=> "ipLoginTemplateId",
			value		=> $self->getSetting("loginTemplateId"),
			namespace => "Auth/IP/Login",
			label		=> "IP Login Template"
		);
		#Return the raw form rows
		return $f->printRowsOnly;
	}
```

After shifting the object off the argument stack and setting the session variable, create a new WebGUI::HTMLForm object. Then, start adding elements of various types to the form. For the ipvisitorId setting, use a WebGUI::Form::User element. For the allowedIpAddress element, use a WebGUI::Form::Text element, and for the templates use a WebGUI::Form::Template element. Also notice that each of the names follows the namespace rule. The name of the authentication plugin is “IP”, thus the ipvisitorId field is named “ipIpvisitorId” and so on. You should also notice that for each of the values you simply use the getSettings method on the name of the setting as it will be referred to throughout the rest of the code base. This makes it less confusing to others reading your code and easier to support for yourself in the future. Finally, return the printRowsOnly method of the WebGUI::HTMLForm object, which will write your form directly to the Authentication tab of the Settings screen within the area for your authentication plugin.

editUserSettingsFormSave()

Now that you've got your form displaying on the Authentication tab of the Settings screen, you need to ensure that your settings are saved to the settings table. Do this by overriding the editUserSetttingsFormSave() method. Once again, this method is empty in the superclass so it is up to you to save your own settings. Like the editUserSettingsForm() method, this method does not need to be callable as it is executed for each authentication plugin installed on your site (see the executing code in WebGUI::Operation::Settings::www_saveSettings()).

To save settings, process the data from the form post, handle any user errors that might exist in the data posted, and then use the WebGUI::Sesssion::Setting API to write your settings to the database. In this example, you have five fields that are posted via the editUserSettingsForm() method to handle.

```
sub editUserSettingsFormSave {
	#shift the object off the argument stack
	my $self	= shift;
	#Create a session variable
	my $session = $self->session;
	#Create a form object variable
	my $form	= $session->form;
	#Create a setting object variable
	my $setting = $session->setting;
	#Create an array for storing errors
	my @errors  = ();
	#Get the data from the form post
	my $ipvisitor		= $form->process("ipIpvisitorId","user");
	my $allowedIPAddress = $form->process("ipAllowedIPAddress","text");
	my $accountTmplId	= $form->process("ipAccountTemplateId","template");
	 my $createAcctTmplId
			= $form->process("ipCreateAccountTemplateId","template");
	my $loginTemplateId  = $form->process("ipLoginTemplateId","template");
	#Write the data to the settings table
	$setting->set("ipIpvisitorId",$ipvisitor);
	$setting->set("ipAllowedIPAddress",$allowedIPAddress);
	$setting->set("ipAccountTemplateId",$accountTmplId);
	$setting->set("ipCreateAccountTemplateId",$createAcctTmplId);
	$setting->set("ipLoginTemplateId",$loginTemplateId);
	#Handle errors
	#Don't allow an IP address to be set if an ipvisitor is not
	if($allowedIPAddress ne "" && $ipvisitor eq "") {
		#Unset the setting.  Don't remove it or it will not be able to be set again
		$setting->set("ipAllowedIPAddress","");
		#Push a message on to the error stack
		push(@errors,"IP Visitor is empty.  Allowed IP Address unset");
	}
	#Return a reference to the error array
	return \@errors;
}
```

After shifting the object off the argument stack and setting up some session variables, begin by processing the form fields from the form post. You'll notice that in this method you must refer to the names of the fields that are posted, which you namespaced in the previous section. Also,

For more information about processing form posts, see the Form Controls chapter.

there is no authentication API method for setting data without using the namespace, so you will simply refer to the full name of the setting throughout this method. If you prefer, you can refer to the full names of the settings throughout the application and avoid the getSetting() method all together, as it is simply a convenience method.

Once you have processed your form variables, write all of the data to the settings table. You may be wondering why errors weren't processed before writing the database settings. Because each authentication method handles writing its own data to the database, you can't stop other data from being posted. Since other settings are going to be saved regardless of any errors that happen in your authentication method, it makes sense to save the settings first and then unset anything that might cause the user problems. You don't want the entire post to fail if one mistake is made. Simply protect the user from harm by unsetting anything that could be a problem, and notify the user that there is an issue with the data submitted.

After writing all of the settings to the database, check for errors. If the user set an allowed IP address, your authentication method is going to try to automatically log in users coming from that IP address as the IP Visitor user. If that user doesn't exist, you have a problem. For that reason, you cannot allow the Allowed IP Address to be set unless the IP Visitor field is also set. Do this by updating the setting in the database to an empty string and pushing an error message onto the error array. It is important not to use the remove method in the settings API as this will completely remove the setting row from the database, making the setting impossible to set again. Finally, return a reference to your error array, which will be reported to the user by the calling function.

User Data

Now that you have your local authentication settings taken care of, you can finally allow user accounts to be created, updated, and deactivated. Before diving into the guts of how to do this, it's important to take a look at user accounts in WebGUI and where data is stored.

All users in WebGUI, regardless of the authentication method, must have a WebGUI user account. As you've seen, even visitors who have not registered for the site have a user account: the visitor account (userId 1). Every user account in WebGUI is stored in the users table which looks like this:

Field	**Type**	**Null**	**Key**
userId	varchar(22)	No	PRIMARY
username	varchar(100)	Yes	UNIQUE
authMethod	varchar(30)	No	
dateCreated	int(11)	No	
lastUpdated	int(11)	No	
karma	int(11)	No	
status	varchar(35)	No	
referringAffiliate	varchar(22)	No	
friendsGroup	varchar(22)	No	

Notice that it is in this table that you store what authMethod (authentication identifier) is used by each user. When a user attempts to log in, you look up the authMethod, create an instance of that method, and call the init method, as seen earlier.

You'll also notice that you don't store a password in this table. That is because the password is strictly dependent on the authentication plugin being used. For instance, the example specifies that passwords be SPA-512 hashed. If the password field existed in the user table and was of length just long enough to support MD5 hashed, the field wouldn't be long enough to hold your password hash. Some authentication methods don't even use passwords that are stored in WebGUI. The LDAP plugin, for instance, authenticates the password posted at login against the LDAP server. The password isn't even stored in WebGUI for that module, so what use would it be in the users table?

Passwords, and other authentication specific user data, are stored in the authentication table, which looks like this:

Field	**Type**	**Null**	**Key**
userId	varchar(22)	No	PRIMARY
authMethod	varchar(30)	No	PRIMARY
fieldName	varchar(128)	No	PRIMARY
fieldData	text	No	

The composite key of userId, authMethod, and fieldName ensures that no user can have more than one data entry for a field within any authentication plugin. It is here that you will store things like each user's password, the last time the user logged in, whether or not the user is allowed to change his/her password, the distinguished name of the user on the LDAP server, and so on.

The authentication superclass contains API methods for retrieving and manipulating this data. One such method was discussed in the login section of this chapter.

$auth->getParams

This method retrieves all of the the fieldName and fieldData pairs for a given user and a given authentication method, returning them as a hash reference.

$auth->saveParams($userId,$authMethod,$propertiesHashRef)

The saveParams method writes all of the key/value pairs passed in as a properties hash reference to the authentication table for the userId and authMethod passed in. The key of the hash is expected to be the fieldName column, and the value of the hash is expected to be the fieldData column.

$auth->deleteParams()

Typically, an authentication method doesn't handle deleting users, so this method, along with the following, should almost never be used; however, it is good to know about them since they do exist and may serve a purpose in any external applications you may write. This method deletes all parameters from the authentication table for the user currently stored in object data. This method can be very dangerous to call as it crosses over authentication methods, and should only be called when completely removing a user from WebGUI.

$auth->deleteSingleParam($userId,$authMethod,$fieldName)

This method deletes a single field from the authentication table for the userId and authMethod passed in. Unlike the authentication settings, it is okay to remove entire rows of data from the authentication table as they are re-inserted by the saveParams() method if the user's information is ever updated.

Finally, it is worth noting one more table where user data is stored, which is the userProfileData table, a sample of which is displayed below:

Field	**Type**	**Null**	**Key**
userId	varchar(22)	No	PRIMARY
email	varchar(255)	No	
firstName	varchar(255)	No	
middleName	varchar(255)	No	
lastName	varchar(255)	No	

This table is an extension of the users table and stores all of the profile data related to a user account. It is flexible and may not look the same on any other WebGUI site, as it can be modified via WebGUI's user profiling interface (for more on user profiling, see the Users chapter of the *WebGUI Administrators Guide*). When creating user accounts, it is sometimes necessary to collect additional data to be stored in this table.

Creating User Accounts

Since each authentication plugin stores different data related to each user, it is the responsibility of the plugin to provide a way for WebGUI's user management system to update this data so users can properly authenticate against the system. Likewise, if you want users to be able to anonymously create accounts using your authentication plugin, it is important to create a way for that to happen as well.

There are two methods available for creating user accounts. The first must always be implemented if you have custom user data that needs to be stored. The second need only be implemented if you wish to allow users to create their own accounts. Please note, however, that there is a global setting for turning anonymous registration on and off, which can lead to some confusion by those using your plugin.

editUserForm()

To allow custom user data to be updated in WebGUI's user management system, you need to provide it the necessary form elements. Do this by overriding the editUserForm() method in the superclass. Again, you will note that this method is empty in the superclass, implying that it will be assumed that your custom authentication method has no custom user data. Also, this method does not need to be callable as it will never be called directly. Instead, each method is executed for each of the authentication plugins installed on your site (the executing code can be found in WebGUI::Operation::User::www_editUser). If you do not override this method, your auth method will appear on the Account tab of the Add/Edit User page of WebGUI's User Management system without any associated form fields.

Like editUserSettingsForm(), override this method by creating a WebGUI::HTMLForm object and return the printRowsOnly method. In the example, you'll want to create a form field for your customized password. It is again vital that you namespace your fields as you do not want the form elements to share the same name as other form elements on the page. Use the namespace rules described above to ensure uniqueness.

```
        sub editUserForm {
                #shift the object off the argument stack
                my $self        = shift;
                #Create a session variable
                my $session = $self->session;
                #Create the WebGUI::HTMLForm object
                my $f           = WebGUI::HTMLForm->new($session);
                #Create the form element for the password field
                $f->password(
                        name=>"ipPassword",
                        label=>"Password",
                        value=>"password"
                );
                #Return the raw form rows
                return $f->printRowsOnly;
        }
```

After shifting the object off the argument stack and setting the session variable, create a new WebGUI::HTMLForm object. Then, add the password element to the form using the WebGUI::Form::Password form module. Notice that the name of the password form element follows the namespace rule.

You might be wondering why the value of password is hard coded to the string “password”. This is for several reasons. First, the password is stored as a ISA-512 hash, so it isn't possible to submit the actual clear text value of the password. Even if you could, you wouldn't want anyone to be able to view source on the page and see the clear text value for each user's password. Instead, put some default text in to “trick” the user into thinking the password is being displayed to avoid an unintentional change. This also serves you on the form post to ensure that the user has actually changed the password to something else before you update.

Finally, return the printRowsOnly method of the WebGUI::HTMLForm object, which will write your form directly to the Add/Edit User screen of WebGUI's User Management System for your authentication plugin.

editUserFormSave()

Now that you've got your form displaying on the Add/Edit User page of WebGUI's User Management System, you need to ensure that the data is saved properly to the authentication table for the correct user. Do this by extending the editUserFormSave() method in the superclass. The superclass method expects a hash reference of properties to write to the authentication table as it simply calls the saveParams() method discussed earlier in the chapter. Like editUserForm(), this method does not need to be callable as it is executed for each authentication plugin installed on your site (see the executing code in WebGUI::Operation::User::www_editUserSave()).

To save settings, process the data from the form post, create the properties hash reference with any data that needs to be saved, and then call the superclass method. If you choose to override this method entirely, you would simply call the saveParams() method rather than calling the superclass method. In this example, you only have your custom password field to handle.

```
sub editUserFormSave {
        #shift the object off the argument stack
        my $self    = shift;
         #Create a session variable
        my $session = $self->session;
        #Create a form object variable
        my $form    = $session->form;
        #Create an empty properties hash reference
        my $props   = {};
        #Get the password from the form post
        my $password = $form->process("ipPassword",”password”);
        #Update the password if it was changed
        if ($password && $password ne "password") {
                $props->{identifier} = Digest::SHA::sha512_base64($password);
        }
         #Call the superclass method with the properties hash ref
        $self->SUPER::editUserFormSave($props);
}
```

After shifting the object off the argument stack and setting up some session variables, create an empty hash reference to store your properties in and then process your password form field. When you processed your settings fields in the editUserSettingsFormSave() method, you were able to handle errors and return messages to the user. In this method, you can still handle errors if you choose to, but you cannot return any messages to the user. For that reason, it is usually a good idea to simply save the data as entered by the user. If you absolutely need validation, it is possible to add client side validation using the WebGUI:HTMLForm API; however, this is not discussed here.

In the example, you simply check to see if a password has been set by the user. Some logic is added to assure that the password is not empty and that the password is not the same as the default value, in which case you are assured that the user has typed something new into this field and you should change the password. Finally, call the superclass method passing in the authentication data to be set.

createAccount()

Now that user data can be created and updated through WebGUI's User Management System, it's time to allow users to create and update their own accounts. As mentioned earlier, you can choose not to allow this to happen; however, there are global settings which enable this ability regardless of what you do. It is usually a good idea to implement these methods and simply return a page stating that the action is not possible with the authentication plugin being used.

The createAccount method is similar to the displayLogin method in that it is almost always called as the visitor user. In order to use the createAccount method for your authentication plugin, you must set it to be the default plugin for the WebGUI site.

If you use the default WebGUI authentication plugin and turn on anonymous registration in the user settings, you will see a “click here to register” link on the default WebGUI login template when you are logged out. Clicking this link brings you to this method which allows you to create your own account for logging in.

This method is really a convenience method which returns a template containing the core form elements that are used for creating a new account. Typically, you will not want to override this method. Instead, you will want to extend it with some options that are available from the superclass.

```
$auth->createAccount([postMethod, templateVariables])
```

“postMethod” is a string which indicates which callable method should be posted to when the user posts the create account form. By default, postMethod is set to “createAccountSave”, a method covered in the next section. If you choose to pass in your own postMethod, the form will be generated so that when the user fills in the account creation form, it will post to whichever method you specify. Keep in mind that this method must be listed in your callable methods. For example, $auth>displayLogin('doCreateAccount') will attempt to post to the doCreateAccount method of your custom authentication plugin.

“templateVariables” is a hash reference containing template variables that you can create in your extension of createAccount that will be exported to the template. By default, the following template variables are available from the createAccount superclass method:

Variable	**Description**
title	Internationalized title of the login page.
create.form.header	<form> open tag containing action and method and hidden fields which specify which page to post to.
create.form.profile	Template loop containing all of the profile fields to display on the page.
profile.formElement	Loop field within create.form.profile which contains the form element.
profile.formElement.label	Loop field within create.form.profile which contains the internationalized label for the form element.
profile.requied	Loop field within create.form.profile which determines whether or not the form field is required for submission.
create.form.profile.<id>.formElement	Individual profile element that can be used outside the form loop where <id> represents the name of the field you wish to access.
create.form.profile.<id>.formElement.label	Individual profile element label that can be used outside the form loop where <id> represents the name of the field you wish to access.
create.form.profile.<id>.required	Individual element that determines whether or not the form field represented by <id> is required or not.

create.form.submit	Submit button for posting the account creation form.
create.form.footer	</form> closing tag
login.url	URL which calls the init method of the default authentication plugin.
login.label	Internationalized text for the login.url field.

You will notice the profile form element loop and single access variables among the template variables. WebGUI allows users to create custom profile fields and determine whether or not those, or any of the default profile fields that come standard, should be displayed or required at login. These loops export that data to the template for display. If you choose to override this method completely, it will be your responsibility to make sure that this works properly.

You extend the template by passing in a hash reference of your own custom template variables that will be used on your create account page. Notice that these template variables do not have a place to enter a username and password since you may require neither for creating an account (you might pull username directly out of an LDAP by asking a user to enter some other information). In order to get those items into your template, you'll need to pass them along as a hash reference to the createAccount superclass method. It is important to note that you cannot change the template variables in the superclass. If you ever want to change something in the superclass variables, you will need to create entirely new template variables and use those instead. You could also choose to override the createAccount method completely and make it do whatever you need it to do.

Extend the createAccount method and add the username and password fields. In addition, ask the user for a password confirmation to validate that the password was entered correctly. Also, if the user is already logged in, you would rather not display this page. Instead, bounce the user to the displayAccount page, which is discussed a bit later. Also, check to make sure that the anonymous registration field is turned on before allowing access to the page. Finally, add an error message that you can display to the user in the case registration fails for some reason.

```
sub createAccount {
        #shift the object off the argument stack
        my $self     = shift;
        #shift off any message that we might have passed in from a failed login
        my $message = shift;
        #Create a session variable
        my $session = $self->session;
        #Create a form object variable
        my $form     = $session->form;
        #Create a setting object variable
        my $setting = $session->setting;
        #Create an empty hash ref to use for our template variables
        my $vars     = {};
        #Return the display account page if the user is already logged in
        if($self->userId ne "1" && $self->userId ne $self->getSetting("ipvisitorId")) {
                return $self->displayAccount($message);
        }
        #Otherwise, if anonymous registration is not enabled, return the login page
        elsif (!$setting->get("anonymousRegistration")) {
                 return $self->displayLogin;
        }
        #Add some template variables to the empty hash we created
        #Return any error messages passed in to the page
        $vars->{'create.message'} = $message if ($message);
        #Create a field for displaying the username
         $vars->{'form.username'}        } = WebGUI::Form::text($session, {
                name  => "username",
                value => $form->process("username"),
         });
        #Create a label field for the username
        $vars->{'form.username.label'  } = "Username";
        #Create a field for displaying the password
        $vars->{'form.password'        } = WebGUI::Form::password($session, {
                 name => "password"
        });
        #Create a label field for the password
        $vars->{'form.password.label'  } = "Password";
        #Create a field for displaying the password confirmation
        $vars->{'form.passConfirm'     } = WebGUI::Form::password($session, {
                name => "passwordConfirm"
        });
        #Create a label field for the password confirmation
        $vars->{'form.passConfirm.label'} = "Password Confirm";
```

```
            return $self->SUPER::createAccount("createAccountSave",$vars);
    }
```

Start out by shifting the session object off the argument stack as you usually do. Then, also shift a message off. Set up some session variables to use through the code and create an empty hash reference to store your template variables. Now, check the userId for the authentication session. If the userId is not “1”, which is the userId of the visitor, and the userId is not that of the ipvisitor (ipvisitors should also be able to create new accounts), that means that the user is already logged in. You don't want users who are already logged in to be able to create new accounts, so show them the displayAccount form instead and pass it any message that might be forwarded. Otherwise, if the anonymous authentication flag is turned off, visitors shouldn't be able to create their own accounts. If this happens, bounce the user to the displayLogin page. If this is the visitor or ipvisitor user, then you will create your custom template variables and pass them to the superclass method.

If you do not make the createAccount method callable, the default template for the Login Box macro will contain links that will not work properly.

createAccountSave()

Now that there is a form available for users to create new accounts, you have to make sure it can be submitted and have the account successfully created. The createAccountSave superclass method handles storing most of the data, but leaves all of the validation up to you. For this reason you will nearly always want to extend or override this method. If you decide to override the method, the createAccountSave method handles:

- creating the new user account
- assigning karma if karma is enabled
- saving all of the profile fields created in the createAccount() superclass method
- saving all the authentication user data
- sending a welcome message if a properly namespaced sendWelcomeMessage field exists for your authentication plugin

- logging in the user
- logging the login
- running any scripts configured to run on registration
- redirecting the user to the page he/she was trying to access when he/she created the account
- and logging the user's registration if the user was invited to join the site.

As you can see, there is a lot to handle, so you will want to extend this method rather than completely override it. You extend the method by passing it the username of the user creating the account, a properties file containing the authentication user data to write to the authentication table, the password of the user creating the account (password is only used to send to the user in the welcome message. If you do not use a password in your authentication scheme, you do not need to pass it in.), and a validated profile data hash, which you will see how to provide shortly.

All validating must be done in in this method. In the example you will need to validate once again that anonymous registration is enabled, that the username entered conforms to WebGUI's username standards, that the password entered matches the confirm password field, and that all of the required profile fields have been filled in.

Once you complete validation, you hand the superclass the data it needs and let it create the user and log him/her in.

```
sub createAccountSave {
    #shift the object off the argument stack
    my $self    = shift;
    #Create a session variable
    my $session = $self->session;
    #Create a form object variable
    my $form    = $session->form;
    #Create a setting object variable
    my $setting = $session->setting;
    #Return the display account page if the user is already logged in
    if($self->userId ne "1" && $self->userId ne $self->getSetting("ipvisitorId")) {
        return $self->displayAccount;
    }
    #Otherwise, if anonymous registration is not enabled, return the login page
```

```
		elsif (!$setting->get("anonymousRegistration")) {
			#Log a security warning to ourselves that someone tried to hack a login
			$session->errorHandler->security("Registration hack attempted!");
			return $self->displayLogin;
		}
		#Process our form fields from createAccount()
		my $username    = $form->process('username',"text");
		my $password    = $form->process('password',"password");
		my $passConfirm = $form->process('passwordConfirm',"password");
		#Use the Profile API to validate the profile data
			my ($profile, $error)
				= WebGUI::Operation::Profile::validateProfileData($session, {
				regOnly => 1
			}
		);
		#Validate the username
		unless ($self->validUsername($username)) {
			$error .= $self->error;
		}
		#Validate that a password was entered
		if($password eq "") {
			$error .= "<li>Password cannot be empty</li>";
		}
		#Validate that the password matches the confirmation
		unless ($password eq $passConfirm) {
			$error .= "<li>Password does not match confirmation</li>";
		}
		#Return the user to the create account page if an error happened
		unless ($error eq "") {
			return $self->createAccount($error);
		}
		#Set the user data properties
		my $props = {};
		$props->{ identifier } = Digest::SHA::sha512_base64($password);
		return $self->SUPER::createAccountSave(
			$username,
			$props,
			$password,
			$profile
		);
	}
```

Start out by shifting the session object off the argument stack as you usually do. Then, set up some session variables to use through the code. Now, check the userId for the authentication session. If the userId is not “1”, which is the userId of the visitor, and the userId is not that of the ipvisitor (ipvisitors should also be able to create new accounts), that means that the user is already logged in. You don't want users who are already logged in to be able to create new accounts, so you'll show them the displayAccount form instead. Otherwise, if the anonymous authentication flag is turned off, and someone tries to post to this form, he is trying to hack your site. If this happens, you will log a security message for yourself and bounce the user to the displayLogin page.

Now, process the form fields that you created in createAccount() for error handling. Use the validateProfileData method of the WebGUI::Operation::Profile API, which will loop through all of the profile fields that are flagged for display on registration, determine whether or not the required fields are filled in, and pass back a hash containing the profile fields to store in the database as well as any errors that may have occurred.

Once you do this, validate the username. There is a convenience method available in the superclass to check the validity of the username supplied as certain rules need to be enforced when creating usernames in WebGUI.

$auth->validUsername($username);

This method security checks the username to make sure that macros are not used, that the username is not a duplicate, that it is not empty, and that it does not contain whitespace at the beginning or end. Any errors are logged in object data and can be retrieved with the error() method.

Finally, validate that the password is not empty and that it matches the password confirmation posted. If any errors occurred in your validation, you return the user to the createAccount method and pass it the errors so they can be displayed.

If all of the data is valid, set a properties hash containing the SHA-512 hashed password and call the superclass method to pass it the appropriate data. This will successfully create the account and log the user in for the first time.

displayAccount()

Once the user is logged in, it is sometimes important that he/she is able to update the authentication information. Do this by overriding or extending the displayAccount method in the superclass. This method is truly a convenience method as it simply sets up some template variables you would otherwise declare yourself. By default, nothing is posted on the displayAccount method, so unlike some of the other methods you've seen, a default post method is not provided and you will have to create your own callable method for posting to should you choose to extend this method.

To extend the method, if you so choose, you will call it with some options that you have seen before.

$auth->displayAccount(postMethod,[templateVariables])

“postMethod” is a string which indicates which callable method should be posted to when the user makes a request. Unlike other similar superclass methods, postMethod is required to extend this. Keep in mind that whatever method you choose here must be listed in your callable methods. For example, $auth->displayAccount('displayAccountSave') attempts to post to the displayAccountSave method of your custom authentication plugin.

“templateVariables” is a hash reference containing template variables that you can create in your extension of displayAccount that will be exported to the template. By default, the following template variables are available from the displayAccount superclass method:

Variable	**Description**
title	The internationalized title of the account page.
account.form.header	<form> open tag containing action and method and hidden fields which specify to which page to post.
account.form.karma	Amount of karma current user has amassed.
account.form.karma.label	The internationalized title for users karma.

account.form.submit	Submit button for posting account information.
account.form.footer	</form> closing tag
account.options	Form loop containing the different display options that are available based on the user's privileges.
options.display	Loop field within account.options which outputs a link to various user account pages within WebGUI.

You extend the template by passing in a hash reference of your own custom template variables that will be used on your display account page. Notice that these template variables do not have any form fields. In order to get the authentication data you would like updated into your template, you'll need to pass them along as a hash reference to the displayAccount superclass method. It is important to note that you cannot change the template variables in the superclass. If you ever need to change something that is in the superclass, you will either need to create entirely new template variables and use those, or override the displayAccount method completely. For this method, either option would be fine.

Let's extend the displayAccount method and allow users to change their passwords. In addition, let's also ask the user to confirm the password to validate that the password was entered correctly. Also, if the user is not logged in, display the displayLogin page instead.

```
sub displayAccount {
                #shift the object off the argument stack
        my $self     = shift;
        #shift off any message that we might have passed in from a failed update
        my $message = shift;
        #Create a session variable
        my $session = $self->session;
        #Create an empty has to use for our template variables
        my $vars     = {};
        #If the user is not logged in, return the displayLogin page
        if($self->userId eq "1" || $self->userId eq $self->getSetting("ipvisitorId")) {
                return $self->displayLogin;
        }
        #Create template varibales
        #Output any message that might have been passed in
        $vars->{'account.message'    } = $message if ($message);
        #Create the password field and label
        $vars->{'form.password'      } = WebGUI::Form::password($session,{
            name  => "password",
            value =>"password"
        });
        $vars->{'form.password.label'} = "Password";
        #Create the password confirm field and label
        $vars->{'form.passwordConfirm'} = WebGUI::Form::password($session,{
            name  => "passwordConfirm",
            value => "password"
        });
        $vars->{'form.passwordConfirm.label'} = "Confirm Password";
         return $self->SUPER::displayAccount("displayAccountSave",$vars);
    }
```

Start out by shifting the session object off the argument stack as you usually do. Then, shift any message off that may have been passed in. Next, set up a session variable to use throughout the code and create an empty hash reference to store your template variables. Now, check the userId for the authentication session. If the userId is “1”, which is the userId of the visitor or that of the ipvisitor (ipvisitors should not be able to update the ipvisitor account), show the user the displayLogin page. If this is a valid registered user, create your template variables for password and password confirm and pass them to the superclass method.

If you do not make the displayAccount method callable, the default template for the Login Box macro will contain links that will not work properly.

As mentioned earlier, you also need to create a custom post method to save the account information, which in this case is simply an updated password. Do this by adding a displayAccountSave method to your authentication plugin.

```
    sub displayAccountSave {
        #shift the object off the argument stack
        my $self    = shift;
        #Create a session variable
        my $session = $self->session;
```

```
		#Create a form object variable
		my $form    = $session->form;
		#If the user is not logged in, return the displayLogin page
		if($self->userId eq "1" || $self->userId eq $self->getSetting("ipvisitorId")) {
			return $self->displayLogin;
		}
		#Get the password data from the form post
		my $password    = $form->process('password');
		my $passConfirm = $form->process('passwordConfirm');
		#Create an empty error message
		my $error       = "";
		#Check to make sure the password is valid
		if($password eq "") {
			$error = "Password cannot be empty";
		}
		elsif($password ne "password" && $password ne $passConfirm) {
			$error = "Password does not match confirmation";
		}
		#If there are no errors and the password was changed, update the password
		if ($error eq "" && $password ne "password") {
			#Save the new password
			my $props   = {};
			$props->{identifier} = Digest::SHA::sha512_base64($password);
			$self->saveParams($self->userId,$self->authMethod,$props);
		}
		#Set a message to return to the user
		my $display = $error || "Account updated!";
		#Return the display account page
		return $self->displayAccount($display);
	}
```

Start out by shifting the session object off the argument stack as you usually do. Then, set up some session variables to use through the code. Now, check the userId for the authentication session. If the userId is “1”, which is the userId of the visitor or that of the ipvisitor (ipvisitors should not be able to update the ipvisitor account), show the user the displayLogin page, as neither of these users should be able to update this account. Next, retrieve password data from the form post and validate that it is not empty and that it matches the password confirm field. If no errors are found and the password was changed (not equal to the default value of “password”), hash the password and save it in the authentication table. Finally, return the user to the displayAccount page with either an error message or a message letting him/her know that the account was updated.

deactivateAccount()

WebGUI's authentication API also gives you the ability to let users deactivate their own accounts. While not a necessary feature, this again is a global authentication setting, so at the very least you should override the method and let users know they aren't allowed to self-deactivate.

This method is a convenience method which returns a template containing the core form elements that are used for deactivating accounts. Unlike other methods that are similar to this, the templateId for this method is hardcoded, so if you want to use your own template you will need to completely override the method.

$auth->deactivateAccount([postMethod])

“postMethod” is a string which indicates which callable method should be posted to when the user posts the create account form. A default post method is not provided by the superclass even though the superclass contains a “deactivateAccountConfirm” method, which is covered in the next section. If you choose to extend this method, you can either pass the deactivateAccountConfirm, which allows you to let the superclass handle, extend yourself, or override. Or, you can pass your own method, in which case the form will be generated so that when the user fills in the account creation form it will post to whichever method you specify. Keep in mind that this method must be listed in your callable methods. For example, $auth->deactivateAccount('deactivateUser') attempts to post to the deactivateUser method of your custom authentication plugin.

Because you cannot pass your own template variables to the method, the template is not extendable. The only reason you would ever extend this method would be to add additional functionality not handled by the superclass. If you do choose to override this method, the superclass method simply checks to make sure the user attempting to use the feature is valid (you cannot deactivate the visitor or admin account) and that the setting to allow users to self-deactivate is enabled.

Let's extend the template and ensure that the IP Visitor account, as well as the regular visitor account, cannot be self-deactivated. If either of these users attempts to deactivate their accounts, you'll return the displayLogin page.

```
sub deactivateAccount {
        #shift the object off the argument stack
        my $self = shift;
        #If the user is not logged in, return the displayLogin page
        if($self->userId eq "1" || $self->userId eq $self->getSetting("ipvisitorId")) {
                return $self->displayLogin;
        }
        #Return the superclass deactivateAccount method
        return $self->SUPER::deactivateAccount("deactivateAccountConfirm");
}
```

Start out by shifting the session object off the argument stack as you usually do. Then, check the userId for the authentication session. If the userId is “1”, which is the userId of the visitor or that of the ipvisitor (ipvisitors should not be able to deactivate the ipvisitor account), show the user the displayLogin page. If this is a valid registered user, allow the superclass to do the rest of the work.

If you do not make the deactivateAccount method callable, the default template for the account options list will contain links that will not work properly.

deactivateAccountConfirm()

As with the deactivateAccount method, the deactivateAccountConfirm() method of the superclass handles most of what you will need to do when deactivating WebGUI accounts. You can choose to override this method completely, but more than likely you'll just need to extend it to add functionality specific to your authentication plugin.

If you do choose to override this method, it simply checks to make sure the user attempting to deactivate the account is not the visitor or the admin, sets the user's account status to “Selfdestructed”, and logs the user out by calling the logout method. It's important to note that if you choose to implement the deactivateAccountConfirm method, that you also have implemented the logout method, otherwise your user will be presented with an error.

Extend the deactivateAccountConfirm method to ensure that the IP Visitor user account cannot be deactivated.

```
sub deactivateAccountConfirm {
        #shift the object off the argument stack
        my $self = shift;
```

```
		#If the user is not logged in, return the displayLogin page
		if($self->userId eq "1" || $self->userId eq $self->getSetting("ipvisitorId")) {
			return $self->displayLogin;
		}
		#Return the superclass method
		return $self->SUPER::deactivateAccountConfirm;
	}
```

After shifting the session object off the argument stack, check the userId for the authentication session. If the userId is "1", which is the userId of the visitor or that of the ipvisitor (ipvisitors should not be able to deactivate the ipvisitor account), show the user the displayLogin page. If this is a valid registered user, allow the superclass to do the rest of the work.

Creating Custom Methods

As shown in the displayAccount section, sometimes you want to create your own custom authentication methods. WebGUI's authentication plugin system is completely extensible, meaning you are not tied down to the methods talked about here so far. In fact, you could go as far as to use only the init() method, and implement your own custom solution.

Typically, however, you will want to use many of the convenience methods available and simply add the extra ones you need. Earlier in the chapter a URL was created which linked to a recoverPassword method. There is no superclass definition for such a method, but it is completely within your power to create one.

To add a custom method to your authentication plugin, simply update the callable stack to add the "recoverPassword" method, and then create the subroutine to do what you define. This example outputs some text which allows the user to send someone on Plain Black's staff an email, but you can imagine what you could do.

```
	sub recoverPassword {
		#shift the object off the argument stack
		my $self = shift;
		#If the user is logged, return the displayAccount page
		if($self->userId ne "1" && $self->userId ne $self->getSetting("ipvisitorId")) {
			return $self->displayAccount;
		}
		#Return a message to the user with instructions for recovering their password
```

```
                my $message = q|
                        To recover your password, please send an email to
                        <a href="mailto:passwordrecovery@plainblack.com">
                                passwordrecovery@plainblack.com
                        </a>.
                        Someone from our staff will send you your new
                        password within 24 hours.
                |;
                return $message
        }
```

Extending Existing Authentication Plugins

Many times you don't need to write a whole new authentication plugin. Sometimes one of the default plugins that comes with WebGUI, or one of the plugins you've created yourself, does almost everything you need, but simply requires a small tweak here or there. In those cases, you can simply extend those plugins directly, instead of WebGUI::Auth, by changing the inheritance statement from use base 'WebGUI::Auth' to use base 'WebGUI::Auth::plugin'.

There are a few caveats to watch out for when doing this. First, remember that all local settings are namespaced according to the name of your plugin. If the calling method uses namespaced settings, you will need to make sure you provide database entries and forms for creating those settings, otherwise you will be stuck with a situation where the superclass is looking for a property that doesn't exist.

Another thing to remember is that the form elements in the editUser and editUserSettings forms all live on the same page. If you plan to run both authentication plugins in the same WebGUI instance, it is vital to override those methods (you can typically copy them exactly) and change the names of the form elements. If you don't do that, you'll wind up with two or more form elements with the same name on the page, and your authentication settings won't be updated properly.

Finally, remember that any time you call the superclass you are not calling the method from WebGUI::Auth (unless the method you are calling is not implemented), but rather the plugin you inherited from. It's easy to get hung up trying to debug an error in the wrong place.

Follow these guidelines and it should be smooth sailing when you write authentication plugins.

Installing Your Authentication Plugin

Now that you know how to write an authentication plugin for WebGUI, you need to know how to install it into the system. While authentication plugins typically don't require their own tables, they do require settings fields to be created, otherwise the state will never hold in the database as the settings API won't save to a field that doesn't exist. Additionally, authentication plugins require templates and need to be added to the config file. You can accomplish this all manually, but it is generally better practice to create an installer and uninstaller for your plugin. How to do each is covered in this section so you have a complete understanding of the integration points and how to create install and uninstall scripts for your plugins.

Manually Installing Your Authentication Plugin

Manually installing an authentication plugin requires five steps:

1. Copy your auth method into the proper folder.
2. Insert any custom settings into the database.
3. Create any template assets associated with your authentication plugin.
4. Add the authentication plugin to WebGUI's configuration file.
5. Configure the authentication settings.

Authentication plugins must be located in /data/WebGUI/lib/WebGUI/Auth/. Ensure that your plugin is copied here before you continue the installation.

For each of the custom settings that you have defined, you will need to create a row in the settings table with some sort of default data. When manually installing, you can leave these as NULL fields and configure the authentication settings via WebGUI's administrative interface. From a MYSQL prompt for the database for the site you are adding your authentication plugin to, type a series of insert statements that look as follows:

```
insert into settings (name, value) values ('settingName','settingValue');
```

Create your template assets by navigating to the import node and creating a new folder somewhere within. From here you will create templates with the proper namespaces using WebGUI's administrative interface. For more information on how to create templates in WebGUI, see the *WebGUI Designers Guide.*

You'll need to edit the WebGUI configuration file for the site in which you'd like to install the plugin, and add the identifier of your plugin to the authMethods key.

```
“authMethods” : [“LDAP”,”WebGUI”,”Yourplugin”]
```

The plugin identifier is the name of your Perl module with the “.pm”. If your module is called IP.pm, then the identifier for your module should be “IP” and so on. Be sure to restart mod_Perl once you have saved the configuration file.

Once you have the configuration file changed properly, your authentication method should be installed. Because this is a manual install, it is important to go to the WebGUI settings in the administrative interface and properly configure the settings for the authentication plugin, otherwise you could have null pointers for templates and other things for which you expect to have data.

Automating Your Authentication Plugin Installation

In reality, if you plan for anyone else to use your authentication method, you are going to need to provide an easy way to install your authentication method. Likewise, if you create this method and then want to install it somewhere else six months later, you will probably forget to do something and it will take extra time to run your installation. For this reason, it's important to create an installer for your authentication plugin that does all of the steps above with just a few easy keystrokes, and as you will see, gives you the power to do even more if you choose.

By creating an installer, you can cut the five steps involved in manually installing your plugin down to a two step process:

1. Copy your auth method into the proper folder.

2. Run the installation script for the authentication method.

Likewise, you can create an uninstaller that will reverse all of the manual steps if you ever want to remove your authentication method.

The install and uninstall scripts should be included at the bottom of your authentication method and should be exported so they can be called on the command line:

```
        use base 'Exporter';
        our @EXPORT = qw(install uninstall);
        use WebGUI::Session;
```

This will allow users to call your install and uninstall methods from /data/WebGUI/lib/ as follows:

```
Perl -MWebGUI::Auth::Myplugin -e install www.mysite.com.conf
Perl -MWebGUI::Auth::Myplugin -e uninstall www.mysite.com.conf
```

Next, create your install method, which is going to relieve most of the tedium involved with the manual install. It will add your authentication plugin to WebGUI's configuration file, it will create your local authentication settings in the database and set some initial values, it will create a folder in the import node in which to store your templates, and then it will proceed to add all of your default templates and commit them.

One other thing that you can do in your install script that you had to do manually (and very well may forget the next time you try to install your authentication plugin) is to create your ipvisitor user so you can set it in your authentication settings, assuring you won't have to remember how or why you do this.

As you can see, all of the guess work has been taken out of the installation. All you need to do now is distribute the file along with very simple instructions for putting the file in the correct folder and running your install script.

```
sub install {
    #First argument on the command line is the config file
    my $config = $ARGV[0];
    #Second argument on the command line is WebGUI's home dir
    #If no second argument exists, use the default home dir
    my $home = $ARGV[1] || "/data/WebGUI";
    #Die if either $home or $config is empty
    unless ($home && $config) {
        die "usage: Perl -MWebGUI::Auth::IP -e install www.example.com.conf\n";
    }
```

```
	#Print out a status message to indicate the install is starting
	print "Installing asset.\n";
	#Open a new WebGUI session for the config file passed in
	my $session = WebGUI::Session->open($home, $config);
	#Add the authentication method to the config file
	$session->config->addToArray("authMethods","IP");
	#Create the ipvisitor user
	my $ipvisitor = WebGUI::User->new($session,"new");
	my $ipvisitorId = $ipvisitor->userId;
	$ipvisitor->username("ipvisitor");
	$ipvisitor->authMethod("IP");
	#Create the local settings
	$session->setting->add("ipIpvisitorId",$ipvisitorId);
	$session->setting->add("ipAllowedIPAddress","");
	$session->setting->add("ipAccountTemplateId","IPtmpl0000000000000001");
	 $session->setting-
>add("ipCreateAccountTemplateId","IPtmpl0000000000000002");
	$session->setting->add("ipLoginTemplateId","IPtmpl0000000000000003");
	# Create a folder asset to store the default template
	# Get the import node
	my $importNode = WebGUI::Asset->getImportNode($session);
	#Add the folder
	my $newFolder = $importNode->addChild({
		className=>"WebGUI::Asset::Wobject::Folder",
		title => "IP Auth Templates",
		menuTitle => "IP Auth Templates",
		url=> "ip_auth_folder",
		groupIdView=>"3"
	},"IPAuthFolder0000000001");
	#Create the templates
	#Account Template Code
	my $accountTmpl = q|
		<h2><tmpl_var title></h2>
		<tmpl_if account.message>
			<tmpl_var account.message>
		</tmpl_if>
		<tmpl_var account.form.header>
		<table>
		<tmpl_if account.form.karma>
		<tr>
			<td class="formDescription" valign="top">
				<tmpl_var account.form.karma.label>
			</td>
```

```
                <td class="tableData">
                        <tmpl_var account.form.karma>
                </td>
        </tr>
        </tmpl_if>
        <tr>
                <td class="formDescription" valign="top">
                        <tmpl_var form.password.label>
                </td>
                <td class="tableData">
                        <tmpl_var form.password>
                </td>
        </tr>
        <tr>
                <td class="formDescription" valign="top">
                        <tmpl_var form.passwordConfirm.label>
                </td>
                <td class="tableData">
                        <tmpl_var form.passwordConfirm>
                </td>
        </tr>
        <tr>
                <td class="formDescription" valign="top"></td>
                <td class="tableData">
                        <tmpl_var account.form.submit>
                </td>
        </tr>
        </table>
        <tmpl_var account.form.footer>

         <div class="accountOptions">
                <ul>
                <tmpl_loop account.options>
                        <li><tmpl_var options.display></li>
                </tmpl_loop>
                </ul>
        </div>
|;
#Create Account Template Code
my $createAcctTmpl = q|
        <h2><tmpl_var title></h2>
        <tmpl_if create.message><tmpl_var create.message></tmpl_if>
        <tmpl_var create.form.header>
```

```
<table>
<tr>
        <td class="formDescription" valign="top">
                <tmpl_var form.username.label>
        </td>
        <td class="tableData">
                <tmpl_var form.username>
        </td>
</tr>
<tr>
        <td class="formDescription" valign="top">
                <tmpl_var form.password.label>
        </td>
        <td class="tableData">
                <tmpl_var form.password>
        </td>
</tr>
<tr>
        <td class="formDescription" valign="top">
                <tmpl_var form.passConfirm.label>
        </td>
        <td class="tableData">
                <tmpl_var form.passConfirm>
        </td>
</tr>
<tmpl_loop create.form.profile>
<tr>
        <td class="formDescription" valign="top">
                <tmpl_var profile.formElement.label>
        </td>
        <td class="tableData">
                <tmpl_var profile.formElement>
        </td>
</tr>
</tmpl_loop>
<tr>
        <td colspan="2">&nbsp;</td>
</tr>
<tr>
                <td class="submitData" colspan="2">
                        <tmpl_var create.form.submit>
                </td>
</tr>
```

```
		</table>
		<tmpl_var create.form.footer>
	|;
	#Login Template Code
	my $loginTmpl = q|
		<h2><tmpl_var title></h2>
		<tmpl_if login.message>
				<tmpl_var login.message>
		</tmpl_if>
		<tmpl_var login.form.header>
		<tmpl_var login.form.hidden>
		<table >
		<tr>
			<td class="formDescription" valign="top">
				<tmpl_var login.form.username.label>
			</td>
			<td class="tableData">
				<tmpl_var login.form.username>
			</td>
		</tr>
		<tr>
			<td class="formDescription" valign="top">
				<tmpl_var login.form.password.label>
			</td>
			<td class="tableData">
				<tmpl_var login.form.password>
			</td>
		</tr>
		<tr>
			<td class="formDescription" valign="top"></td>
			<td class="tableData">
				<tmpl_var login.form.submit>
			</td>
		</tr>
		</table>
		<tmpl_var login.form.footer>

		 <div class="accountOptions">
			<ul>
				<li>
					<a href="<tmpl_var recoverPassword.url>">
						<tmpl_var recoverPassword.label>
					</a>
```

```
                                </li>
                                <tmpl_if anonymousRegistration.isAllowed>
                                <li>
                                        <a href="<tmpl_var createAccount.url>">
                                                <tmpl_var createAccount.label>
                                        </a>
                                </li>
                                </tmpl_if>
                        </ul>
                </div>
        |;
        #Add the templates to the folder
        $newFolder->addChild({
                className               =>"WebGUI::Asset::Template",
                ownerUserId             =>'3',
                groupIdView             =>'7',
                groupIdEdit             =>'12',
                title           =>"IP Auth Account Template",
                menuTitle       =>"IP Auth Account Template",
                url             =>"ip_auth_account",
                namespace               =>"Auth/IP/Account",
                template        =>$accountTmpl,
                }, 'IPtmpl0000000000000001'
        );
    $newFolder->addChild({
                className       =>"WebGUI::Asset::Template",
                ownerUserId             =>'3',
                groupIdView=>'7',
                groupIdEdit     =>'12',
                title           =>"IP Auth Create Account",
                menuTitle       =>"IP Auth Create Account",
                url             =>"ip_auth_create",
                namespace       =>"Auth/IP/Create",
                template        =>$createAcctTmpl,
                },'IPtmpl0000000000000002'
        );

        $newFolder->addChild({
                className=>"WebGUI::Asset::Template",
                ownerUserId=>'3',
                groupIdView=>'7',
                groupIdEdit=>'12',
                title=>"IP Auth Login",
                menuTitle=>"IP Auth Login",
                url=>"ip_auth_login",
                namespace=>"Auth/IP/Login",
                template=>$loginTmpl,
                },'IPtmpl0000000000000003'
        );
        #Commit the version tag
        my $workingVersionId = WebGUI::VersionTag->getWorking($session)->getId;
        my $tag = WebGUI::VersionTag->new($session,$workingVersionId);
        if (defined $tag) {
                print "Committing tag\n";
                        $tag->set({comments=>"IP Auth Install"});
                $tag->requestCommit;
        }
        #Close the WebGUI Session and finish
        $session->var->end;
        $session->close;
        print "Done. Please restart Apache.\n";
}
```

Begin by accepting the config file and home directory as arguments from the command line. If no home directory is supplied, assume it's the default home directory for a WebGUI install. Then, create a WebGUI session using the WebGUI::Session API. Once you have a WebGUI session open, you can start using all of the WebGUI API's to install your authentication plugin. Use the WebGUI::Session::Config API to add your plugin to the config file. Then, use the WebGUI::User API to create your IP Visitor User. You then use the WebGUI::Session::Setting API to add your local authentication settings to the database with default values including the userId of the IP Visitor User you just created.

Next, get the import node with the WebGUI::Asset API and add a folder to it by calling the addChild method. Proceed to add the templates to the folder in this same manner.

Finally, get the current working version tag and commit it so your templates are fully installed and ready for use. Close the session, and return a message to the user letting him/her know that he/she should restart Apache as you have updated WebGUI's configuration file.

Now that you have an automated installation method, you should also automate uninstalling your authentication plugin. The steps needed include:

1. Remove all your custom settings from the database.
2. Remove all your templates from the system.
3. Remove the folder you created from the system.
4. Remove your plugin from the configuration file.
5. Remove the ipvisitor user.
6. Handle all of the users that are currently using your IP auth module

To do all of this manually could be quite a lengthy job if you had thousands of users on your site using the IP method. However, you can automate all of this, and add a command line script similar to your installation file.

```
sub uninstall {
	#First argument on the command line is the config file
	my $config = $ARGV[0];
	#Second argument on the command line is WebGUI's home dir
	#If no second argument exists, use the default home dir
	my $home = $ARGV[1] || "/data/WebGUI";
	#Die if either $home or $config is empty
	unless ($home && $config) {
		die "usage: Perl -MWebGUI::Auth::IP -e uninstall www.example.com.conf\n";
	}
	#Print out a status message to indicate the uninstall is starting
	print "Uninstalling asset.\n";
	#Open a new WebGUI session for the config file passed in
	my $session = WebGUI::Session->open($home, $config);
	#Remove the authentication method from the config file
	$session->config->deleteFromArray("authMethods","IP");
	#Delete all of the authentication settings
	$session->setting->remove("ipIpvisitorId");
	$session->setting->remove("ipAllowedIPAddress");
	$session->setting->remove("ipAccountTemplateId");
	$session->setting->remove("ipCreateAccountTemplateId");
	$session->setting->remove("ipLoginTemplateId");
	#Reset the default IP setting if it is set to ours
	if($session->setting->get("authMethod") eq "IP") {
		$session->setting->set("authMethod","WebGUI");
	}
	#Get all the users that are using our authentication method
```

```
my $sql = q|
	SELECT
		userId
	FROM
		users
	WHERE
		authMethod = 'IP'
|;
my @users = $session->db->buildArray($sql);
#Delete all of the users using our authentication method
foreach my $userId (@users) {
	my $user = WebGUI::User->new($session,$userId);
	if(defined $user) {
		$user->delete
	}
}
#Clean up any of the remaining IP fields in the authentication table
$sql = q|
	DELETE
	FROM
		authentication
	WHERE
		authMethod='IP'
|;
$session->db->write($sql);
#Get all of the templates that are in our namespace
$sql = q|
	SELECT
		assetId
	FROM
		template
	WHERE
		namespace IN ('Auth/IP/Create','Auth/IP/Account','Auth/IP/Login')
|;
my @assets = $session->db->buildArray($sql);
#Push the assetId of the folder onto the asset array
push(@assets,"IPAuthFolder0000000001");
#Purge all of the templates and the folder
foreach my $assetId (@assets) {
	#Instantiate the asset
		my $asset = WebGUI::Asset->newByDynamicClass($session,$assetId);
	#Purge the template if it is defined
	if(defined $asset) {
```

```
                        $asset->purge;
                }
        }
        #Close the session and finish
        $session->var->end;
        $session->close;
        print "Done. Please restart Apache.\n";
}
```

Begin by accepting the config file and home directory as arguments from the command line and creating the WebGUI session as you did in your install method. Then, use the WebGUI::Session::Config API to remove your plugin from the config file. Use the WebGUI::Session::Setting API to remove your local authentication settings from the database and make sure that the default authMethod setting is not your authentication method. If it is, set it back to the WebGUI authentication method.

Next, find all the users that currently use your IP auth method, which includes your IP Visitor user. If you were so inclined at this point, you could change all of the users to use a different authMethod, but for the sake of this example simply delete all of these users with the WebGUI::User API. Once you've done this, clean up any fields that might still exist in the database that reference your authMethod (users that may have once used your auth method, but no longer do, might still have properties configured).

Now, find all of the templates in your namespace from the template table, push on the assetId of the folder from the install script, and proceed to purge your templates and folders.

Once this is complete, your asset is uninstalled. Close the WebGUI session and alert the user that Apache must be restarted in order for the changes to take affect. You now know everything you need to create powerful authentication plugins to run your WebGUI site as you see fit.

Workflow Activities

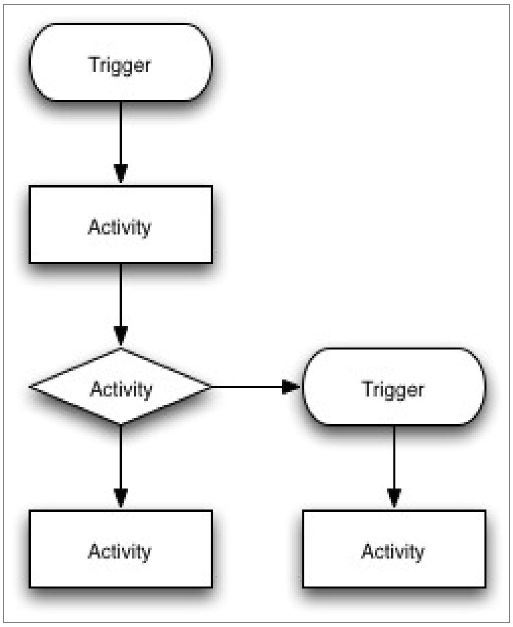
Workflow Activities are plugin points into the WebGUI workflow engine. They allow you to do offline or asynchronous tasks, as well as a few online or synchronous tasks. Examples of workflow activities include asking a user a question, sending an email, publishing some content, deleting temporary files, and running external programs.

The Workflow Engine

To understand how to write good workflow activities, you really need to understand how the WebGUI workflow engine works. A workflow engine is an event-triggered state machine and execution system. In plain words, something happens to cause a workflow to be executed, the workflow engine keeps track of where it is in the process, and runs each task along the way.

Steps of WebGUI Workflow

Everything starts with a trigger: something must kick off the workflow. It could be that it's noon on Saturday, or that a user published some content, or that a user registered on the site, or that you received some email in your inbox. You can even have one workflow be the trigger of another, as shown in the diagram to the right.



Then you have the workflow itself. A workflow is a list of activities (or tasks) to be executed in a particular order. A workflow can be a single activity, or many of them strung together.

An activity is a task, or a single atomic unit of work, that you need to be done. It could be delete an asset, roll back a version tag, email a user, run an external program, check a folder for a file, etc.

Synchronicity

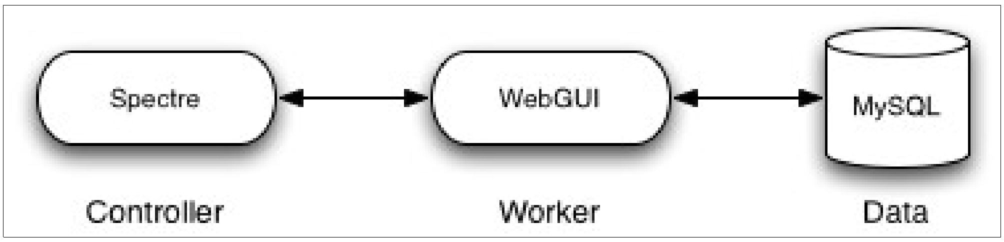
WebGUI's workflow engine is both synchronous and asynchronous. Most workflow systems are one or the other.

Synchronous (in the user interface called “realtime”) workflows execute to completion immediately upon being triggered. Because of this, there can't be anything in a realtime workflow that could get stuck waiting for external input, or that would take a long time to execute. For example, if a workflow is set up to be realtime, and you have an approval process in the workflow, the user will have to wait hours, or even days, for the page to return after clicking on a link. In reality, what would happen is that the page would time out and the user would get an error. Obviously, this is unacceptable, so realtime workflows should only be used in situations where you aren't blocking for input, and where things can happen very quickly.

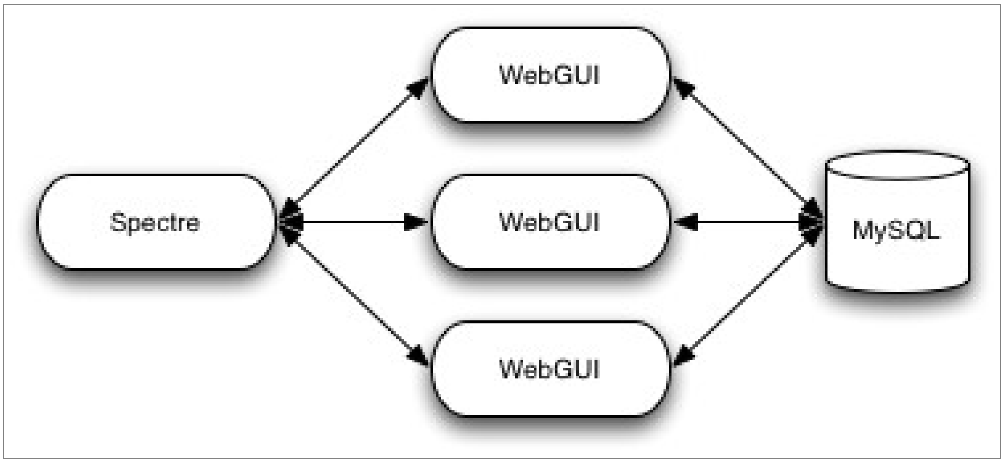
Asynchronous workflows, on the other hand, happen in the background. They can take all the time they need to execute, because no user is immediately waiting on the result. In asynchronous workflows, there is a controller, also called a governor, that executes one task in a workflow, then another task perhaps in a completely different workflow, and so on. The controller is in charge of what gets executed and when. Because of this, there can be priorities assigned to workflows, and the system can queue many workflows, and be able to handle many more workflows than could otherwise be handled. It can take advantage of queuing to execute workflows over a longer period of time. For example, if the system had 10,000 workflows start all at once, it's likely that your server would not be able to handle them and crash. However, an asynchronous system just puts all the workflows into a queue, prioritizes them, and slowly begins its work of executing all 10,000 workflows.

Workflow Engine Components

The workflow engine consists of a controller that hands out tasks, a worker that executes the tasks, and some data to operate on.



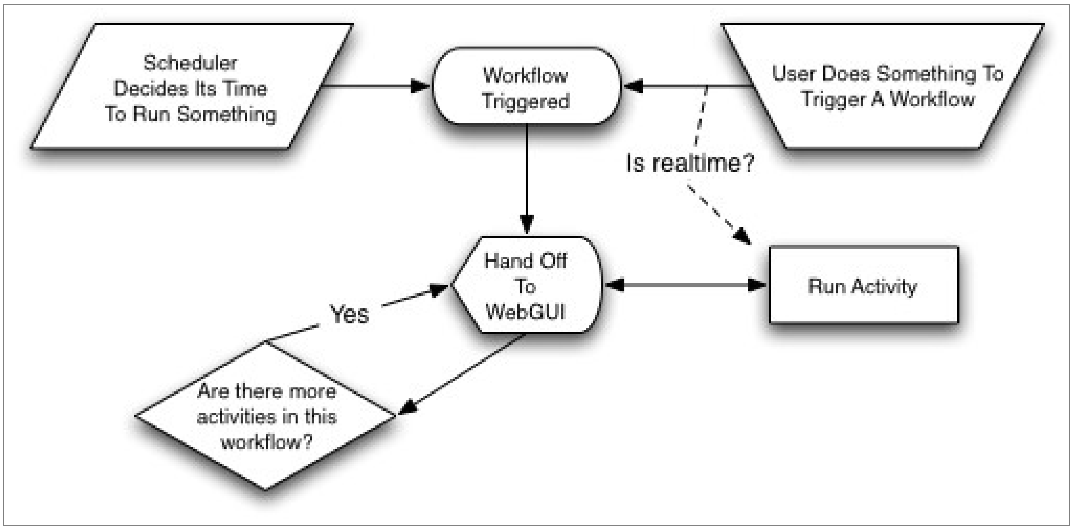
The controller is called Spectre. It keeps a list of what workflows need to be executed and at what priority. When it's time to execute a workflow, it tells WebGUI to actually do the work. WebGUI is the worker because it already has all the code loaded and waiting to run, so why write/run another worker? It has another advantage, though. Since WebGUI can be load balanced, it makes the workflow engine infinitely scalable.



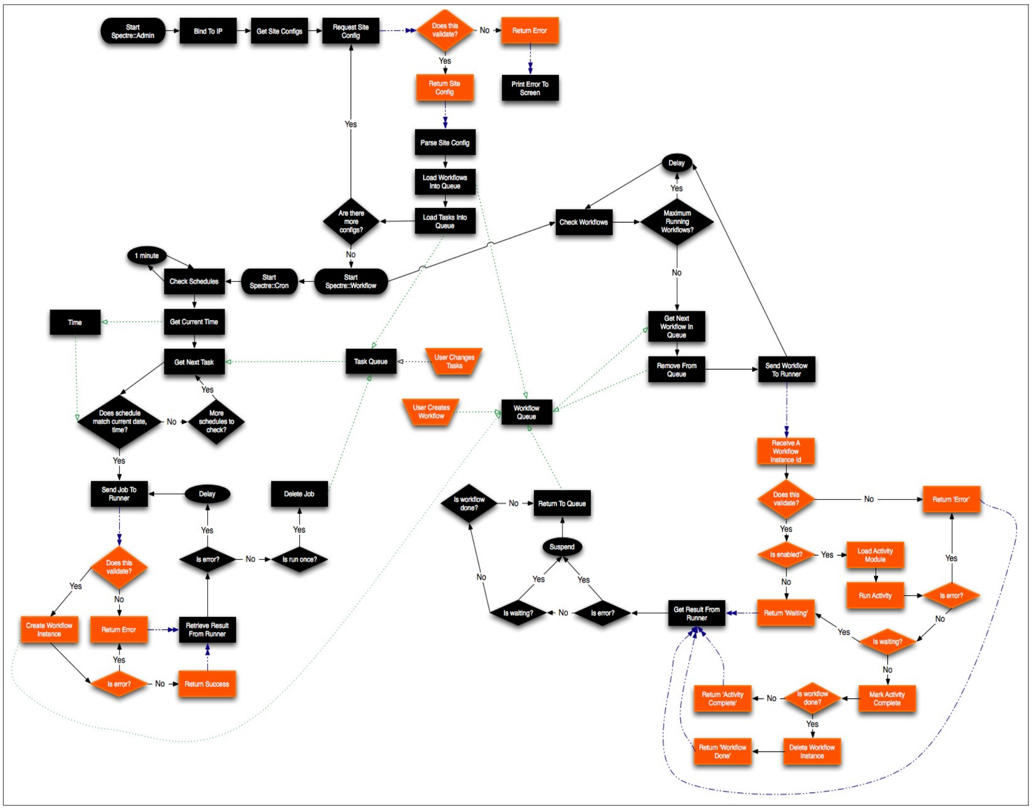
Spectre and WebGUI work in conjunction like this to accomplish all their tasks.

Spectre and WebGUI Communicate

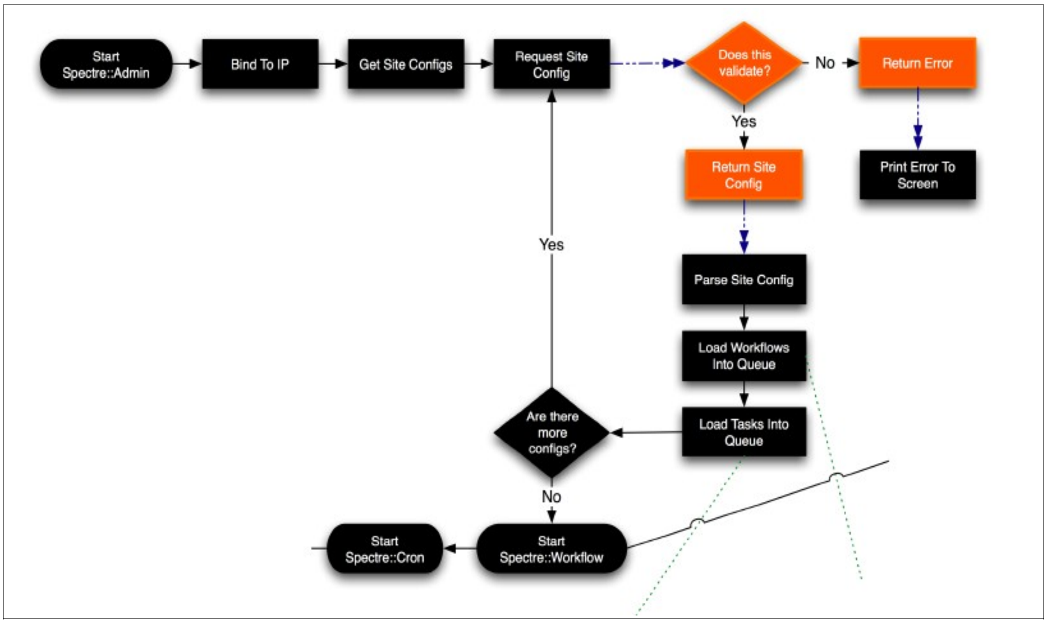
Spectre and WebGUI talk to each other. WebGUI lets Spectre know when a new workflow has been triggered by a user or some other mechanism. Likewise, it lets Spectre know when priorities change, or when a user modifies a workflow. In turn, Spectre tells WebGUI when it's time to run a workflow, or when something has been triggered based upon time or date. From a high level, the communication looks like this:


Scheduler Decides Its Time To Run Something
Workflow Triggered
User Does Something To Trigger A Workflow
Is realtime?
Hand Off To WebGUI
Run Activity
Yes
Are there more activities in this workflow?

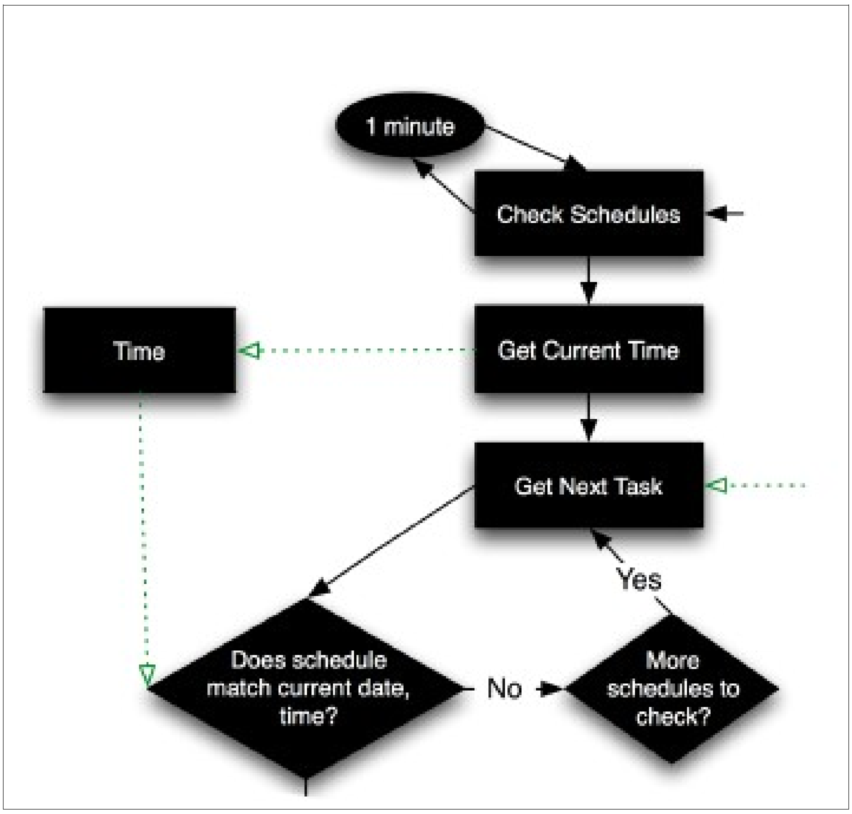
A more detailed view of the interaction between Spectre (black) and WebGUI (orange) looks like this:



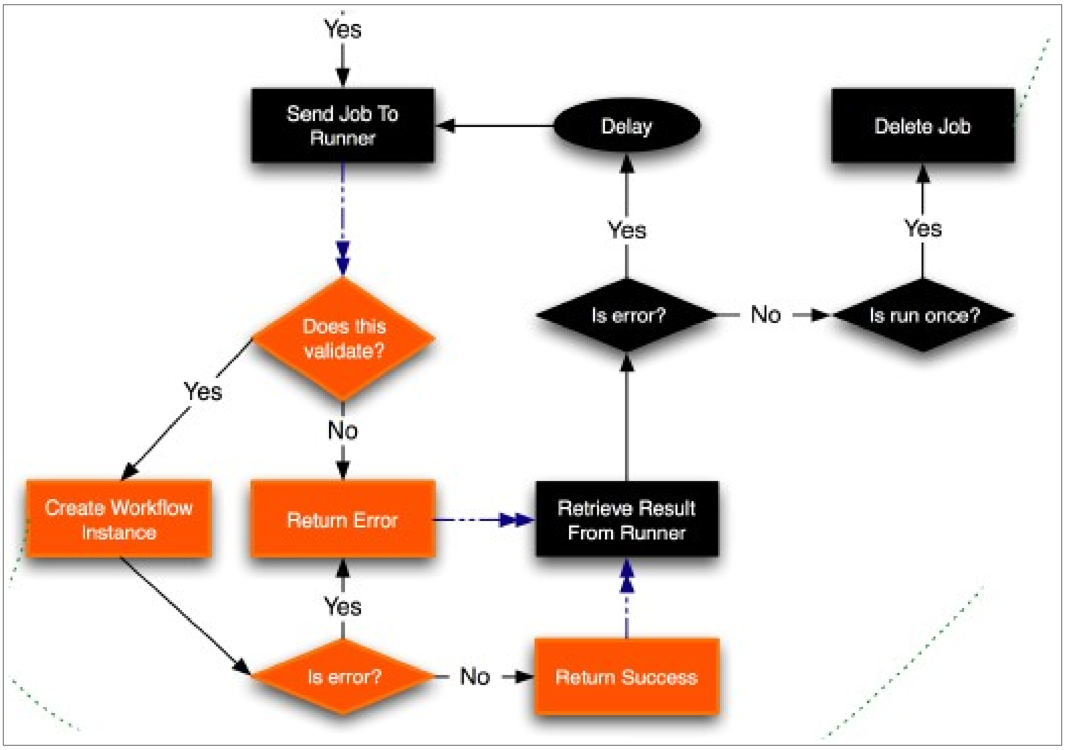
Let's zoom in a little on that. First, Spectre starts up and asks WebGUI what workflows and cron jobs (events triggered by time) are pending in the queue for each site. Then it starts its workflow and cron job handlers.



Upon starting the cron handler, it starts checking to see if any cron tasks are ready to be triggered. That process looks like this:

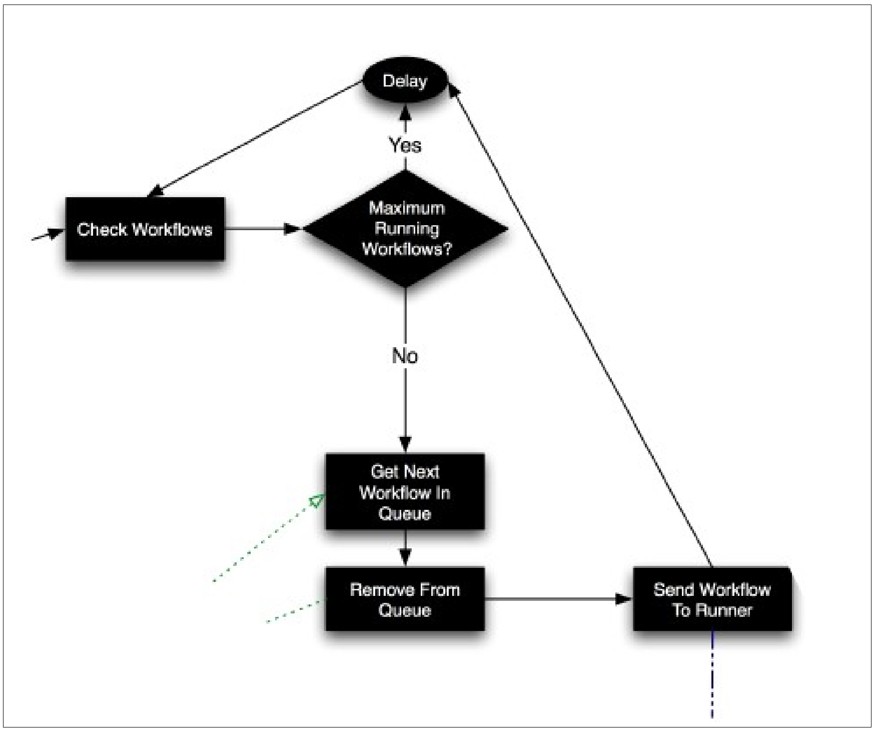


And if something is ready to be triggered, then it tells WebGUI that it's time to trigger that event.

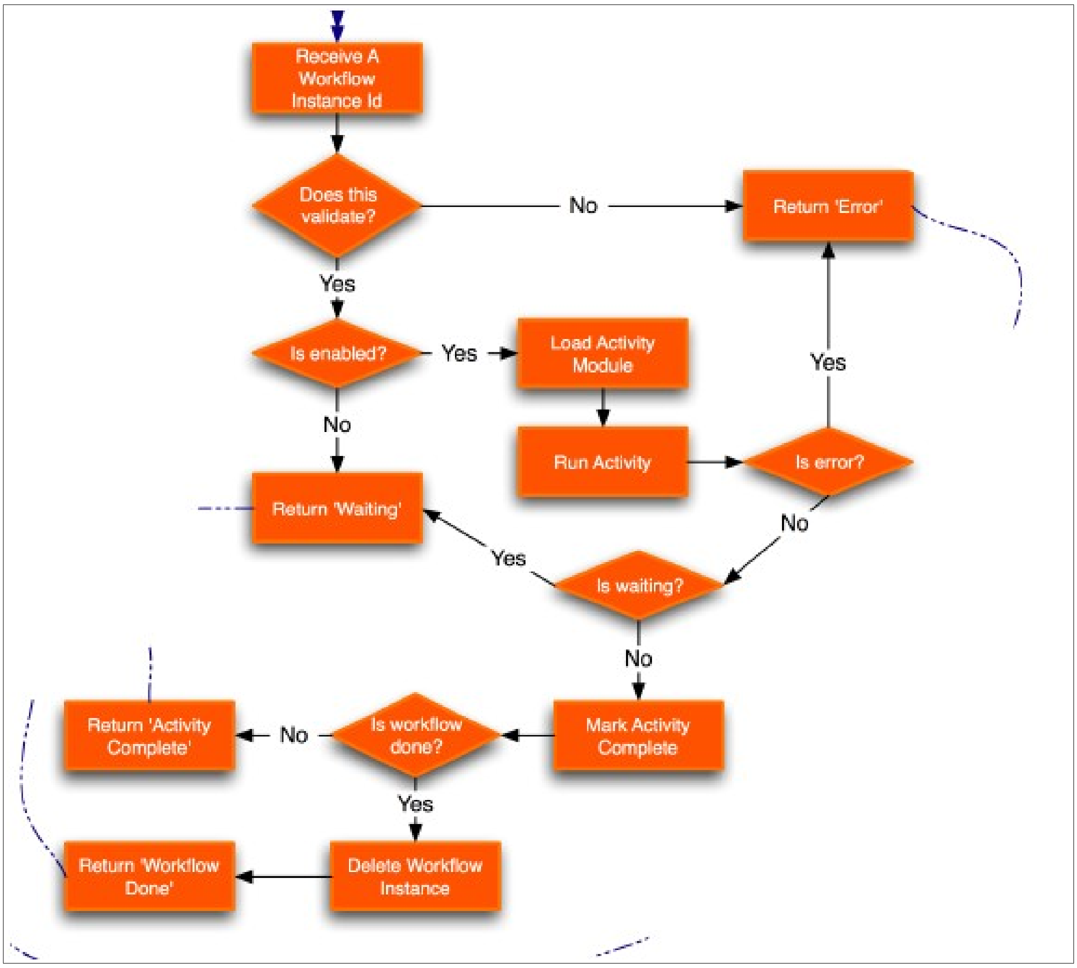


At this point WebGUI has to do a little sanity check to make sure that it actually is Spectre making the request, and that the workflow being requested to be triggered exists. If it does, it creates a new workflow instance (a running workflow), which is discussed next.

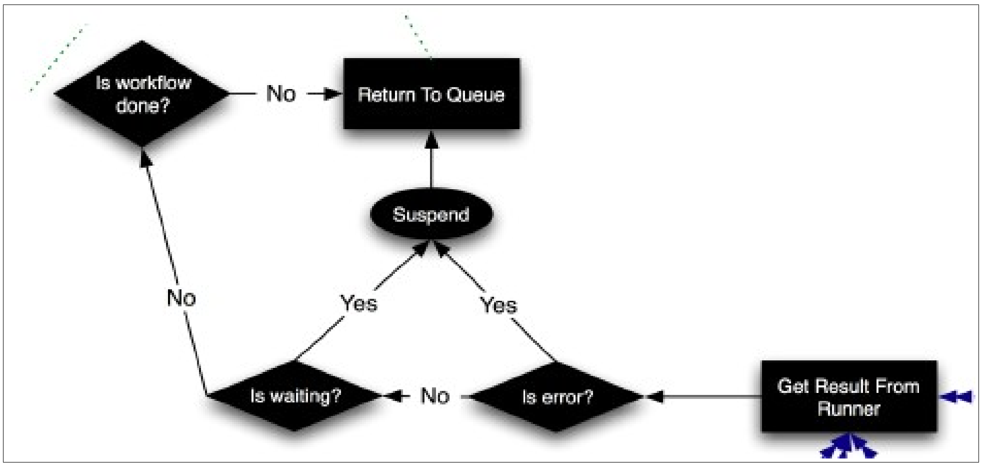
Once Spectre has started its workflow handler, it can start running through its queue to get the next workflow to be run. It also has to keep track of how many workflows it already has running that haven't completed so it doesn't overwhelm WebGUI with a lot of extra requests.



Then, it can tell WebGUI that it's time to run a particular workflow. It does the same kind of validity handshake that it did with the scheduler.



Then, WebGUI determines what the next activity to run is, loads the module for that activity, and runs it. Depending upon whether it's the last activity in the workflow or not, and whether it executed successfully, it will tell Spectre if it completed the activity, if it's waiting for input, if it had an error, or if the workflow is done.



WebGUI::Workflow::Activity API

Though there are many methods provided by the WebGUI::Workflow::Activity class, there are five methods that you absolutely must know about in order to write a workflow activity. They are definition(), execute(), and the three state methods ERROR(), WAITING(), and COMPLETE(). Look at the API for the version of WebGUI you're working with for full API details.

definition()

If you're familiar with writing assets or form controls, then the definition method should be familiar, because it works in much the same way. The definition method tells the activity what data it consumes, and sets a few properties about the activity. Here's an example from an existing activity:

```
sub definition {
   my $class = shift;
   my $session = shift;
   my $definition = shift;
   my $i18n = WebGUI::International->new($session,
"Workflow_Activity_CleanTempStorage");
   push(@{$definition}, {
      name=>$i18n->get("activityName"),
      properties=> {
         storageTimeout => {
            fieldType=>"interval",
            label=>$i18n->get("storage timeout"),
            defaultValue=>6*60*60,
            hoverHelp=>$i18n->get("storage timeout help")
            }
        }
      });
   return $class->SUPER::definition($session,$definition);
}
```

As you can see, you set the human readable name of the activity (in this case using internationalization), and set what fields should be displayed to the user as settings.

States

When developing your activity, you need a way to communicate to the workflow engine whether or not your activity successfully completed. That's where states come in. There are currently three states that you can use with your activity, and they are called as methods that you would return.

return $self->COMPLETE;

Complete is used to let the workflow engine know that you have completed your unit of work successfully and it's okay to move on to the next step in the workflow.

return $self->ERROR;

The error state means something went wrong. It could be that you tried to connect to a mail server and it was down. The workflow engine doesn't need to know why your activity failed, only that it did. However, the human system administrator does, so in the case of an error, put something in the log.

return $self->WAITING;

Waiting lets the workflow engine know that you didn't encounter an error, but the activity didn't complete just the same. This can happen if you're waiting for a user to give you some input, or perhaps whatever the activity is doing (maybe sending out 100,000 emails) is taking a really long time. Therefore, you need to give the resources back to the workflow engine, and it will call this activity again when there's time.

execute()

The execute method is the equivalent of the view method in an asset. It's the work that's going to get done. This is where you put your working code, and also where you use the state methods.

The execute method gets three objects passed into it. As with any object, the first object passed in is a reference to the activity itself. With that you can get access to the session, the states, properties set by the workflow editor, etc. The second object is a reference to the data this activity will be operating on. Some activities don't work on an object, but most do. So it might be a WebGUI::Group, a WebGUI::VersionTag, a WebGUI::User, or some other object. The third, and final, object passed in is a reference to the running workflow; this is the instance object.

Here's a typical execute method from an existing activity:

```
sub execute {
   my $self = shift;
   my $versionTag = shift;
   my $completion = $versionTag->commit({timeout=>55});
   if ($completion == 1) {
      return $self->COMPLETE;
   } elsif ($completion == 2) {
      return $self->WAITING;
   }
   return $self->ERROR;
}
```

WebGUI::Workflow::Instance API

There is generally not a lot that you need to use out of the instance API except for one category of methods: scratch. Scratch methods allow you to attach data to the workflow instance. This can be useful for a couple of things.

The first, and most used, is to maintain state information. For example, let's say your workflow activity sends out 100,000 emails, and that your mail server is capable of accepting a maximum of 10 emails per second. After 60 seconds your workflow activity should give up, and return the resources back to the workflow engine (more on this later in the chapter), so that means you will have only sent out 600 emails. You can set a scratch variable in the workflow instance to remind your activity where it left off the last time it ran.

The second is to pass information between two or more workflow activities within a workflow. Since the workflow activities run independently, there's no way for them to pass data from one to the next, at least directly. By setting a scratch variable in one activity, another down the line can read that scratch variable and make use of its data. You must be careful when using scratch variables for this reason. If you set a scratch variable that you don't intend for another workflow activity to use later, then you should delete it when you no longer need it.

Scratch variables are automatically partitioned between workflow instances, so you don't have to worry about one workflow instance using another instance's scratch variables. Likewise, when a workflow instance is complete (when it has no more activities to run) it will automatically clean up any outstanding scratch variables.

setScratch()

Use the setScratch method to set a variable. The first parameter is the name of the variable, and the second is the value. Note that scratch variables can only be scalars, not references, hashes, arrays, or objects. Therefore, if you need to store a hash, for example, you should serialize it into a scalar using JSON. Here's an example use of setScratch using JSON:

```
$instance->setScratch( “cars”, JSON::to_json(\%cars) );
```

getScratch()

Use the getScratch method to retrieve a variable. Its only parameter is the name of the variable you wish to retrieve. Here's an example, again using JSON to deserialize a hash:

```
my $cars = JSON::from_json( $instance->getScratch(“cars”) );
```

deleteScratch()

Use the deleteScratch method to delete a variable. Its only parameter is the name of the variable you wish to delete. Here's an example:

```
$instance->deleteScratch(“cars”);
```

Rules to Work By

There are a few things to keep in mind as you design your activities. Let's call them rules to work by.

First, no workflow activity should run for longer than 60 seconds at a time. There are two practical reasons for this. One is that workflows are handled between WebGUI and Spectre talking through Apache. Therefore, if it takes longer than 60 seconds, the connection might time out. The other is that workflow is about cooperative multitasking. That means that processes should give up their resources once in a while in case there is something higher priority to get done. Without this, the workflow queue could become bogged down very quickly on busy and complex sites. If you need something to run longer than 60 seconds, then either maintain state using scratch variables so you can pick up where you left off, or hand off the task to a daemon that is designed to handle long running processes.

Second, clean up after yourself. Don't leave temporary files, scratch variables, and other data lying around. If you make a mess, clean it up so not to gum up the works.

Third, make your activities small and uncomplicated. Rather than making a really complicated activity, break up the activity into small parts. This will make coding and testing easier, and will allow your users to extend the uses of your activities in ways you never dreamed of.

Workflow Activity Examples

Here are a few workflow activity examples to get you started.

Hello World

The following example demonstrates creating a bare bones workflow activity that simply prints “Hello World” to a file. Print it to a file because workflow activities have no human viewable output.

```
package WebGUI::Workflow::Activity::HelloWorld;

use strict;
use base 'WebGUI::Workflow::Activity';

sub definition {
   my ($class, $session, $definition) = @_;
   push(@{$definition}, {
      name          => 'Hello World',
      properties   =>  { }
      });
   return $class->SUPER::definition($session,$definition);
}

sub execute {
   my ($self, $object, $instance) = @_;
   open my $file, '>', '/tmp/helloworld.txt';
   print {$file} "Hello World";
   close($file);
   return $self->COMPLETE;
}
```

Email Poll Example

Let's say that your boss tells you that she wants to create a poll, but instead of putting it out on the web site, she wants to conduct the poll via email. “Great!” you say, and think to yourself, “How the hell am I going to do that?”

Functional Specification

Your boss has given you instructions on how her email poll should work.

1. It should email a group of users from the web site.
2. The email should contain the question, and a list of possible answers as links.
3. The users should be able to click on the links to respond to the poll.
4. The responses should be stored somewhere so reports can be generated later.
5. The poll should end after one month and reject further responses after that time.

Technical Specification

From your boss's list of features, you can draft a simple technical specification.

1. The workflow type for this activity should be WebGUI::Group, so that the group object passed into the activity will be the list of users emailed.

2. The email will have to be HTML based so you can put links in it.

3. You can use WebGUI's built-in send email to group feature in WebGUI::Mail::Send to email the group.

4. You'll have to create some www_ methods in the activity to handle the responses when users click on a link.

5. The workflow activity should return WAITING during the response period so that it can collect responses.

Ideally, at this point you'd also create a database schema for storing the data, a flow chart diagram illustrating the use case, and some sample screen shots of what the email and response pages would look like. Then have your boss sign off on these things before proceeding with development.

The Code

Now that your boss has signed off, you can begin coding.

Internationalization, heavy comments, and POD have been left out for the sake of clarity in this example, but when you're writing your own activities you should always use them.

Always start writing an activity by using the code skeleton provided in lib/WebGUI/Workflow/Activity/_activity.skeleton. Begin with the typical header information.

```
package WebGUI::Workflow::Activity::EmailPoll;
use strict;
use base 'WebGUI::Workflow::Activity';
use WebGUI::Mail::Send;
```

Now, you need to set up your definition. Call the activity something simple yet meaningful: “Email Poll”. Then, define a few properties about the activity that you expect your workflow editor to set. You need to have a subject for the email, the question, and the list of answers. Those are all pretty obvious, but because you know your boss likes to change her mind a lot, also build in a timeout setting, so that if she decides that the poll should last for two weeks or two months instead of four weeks, it's an easy change.

```
sub definition {
   my ($class, $session, $definition) = @_;
   push(@{$definition}, {
      name=>"Email Poll",
      properties=> {
         subject => {
            fieldType=>"text",
            label=>"Subject",
            hoverHelp=>"Put the email subject here."
            },
         question => {
            fieldType=>"HTMLArea",
            label=>"Question",
            hoverHelp=>"Put your question/description here."
            },
         answers => {
            fieldType=>"textarea",
            label=>"Answers",
            hoverHelp=>"Put your answers here. One per line."
            },
         timeout => {
            fieldType=>"interval",
            label=>"Timeout",
            defaultValue=>60*60*24*7,
            hoverHelp=>"How long should we allow people to respond?"
            },
         }
      });
   return $class->SUPER::definition($session,$definition);
}
```

Next, create your execute method, which does the work. Because this activity contains a multi-step process, use the execute method as a “main” or controller. You could also break up these steps into separate activities, but that might be too complicated for your first example, so this example keeps them in one activity. This example makes use of scratch variables to maintain the state of which step you're on.

```
sub execute {
   my ($self, $group, $instance) = @_;
   my $endDate = $instance->getScratch("pollEndDate");
   # haven't sent the email yet
   if ($endDate eq "") {
      $self->sendMessage($group);
      $instance->setScratch("pollEndDate", time() + $self->get("timeout"));
      return $self->WAITING;
   }
   # time isn't up
   elsif (time() < $endDate) {
      return $self->WAITING;
   }
   # time is up
   else {
      $instance->deleteScratch("pollEndDate");
      return $self->COMPLETE;
   }
}
```

Because you defined a sendMessage method in the execute method, you need to define that in your code. Here, you're going to make use of the WebGUI::Mail::Send package to send out your emails to the group. That way you don't actually have to look at the group, find the users, look up their email addresses, and send out a number of individual emails. In the email, define URL's for the responses, and one of those parameters is the “method” name that you're going to use to handle those responses. That method is built next.

```
sub sendMessage {
   my ($self, $group) = @_;
   my $url = $self->session->url;
   my $mail = WebGUI::Mail::Send->create($self->session, {
      toGroup => $group->groupId,
      subject => $self->get("subject"),
        });
    my $message = $self->get("question").'<ul>';
    foreach my $answer (split("\n", $self->get("answers"))) {
        next if $answer =~ m/^\s*$/;
        my $url = $url->page(
            "op=activityHelper;class=EmailPoll;method=respond;response="
                .$url->escape($answer)
                .";instanceId=".$self->getId,
            1
            );
        $message .= qq|<li><a href="$url">$answer</a></li>|;
    }
    $message .= '</ul>';
    $mail->addHtml($message);
    $mail->queue;
}
```

Now, you can build the handler to deal with users clicking on the response links in their emails. Note that if the workflow instance isn't still valid, then the user won't be able to vote and will get a friendly rejection message.

```
sub www_respond {
    my $session = shift;
    my $instance = WebGUI::Workflow::Instance->new(
                $session,
                $session->form->get("instanceId")
                );
    my $output = q|<h1>Thank You</h1>|;
    if (defined $instance) {
        my $response = $session->form->process("response", "text");
        # store the response somewhere
        $session->db->setRow("EmailPollResponses", "instanceId", {
            instanceId  => $instance->getId,
            response    => $response
            });
        $output = q|Thank you for your response.|;
    }
    else {
        $output = q|Thank you for your response, but unfortunately our poll has ended.|;
    }
    return $session->style->userStyle($output);
}
```

And that's all. With only four subroutines, you've built an email based polling system that can be tied into WebGUI.

Configuration

Now that you've built your activity, you simply need to put it in the WebGUI config file so that your users can use it. This is fairly simple. Just add the workflow object type of “WebGUI::Group” to the “workflowActivities” directive in the config file. Then, put your activity in that new object type.

```
"workflowActivities" : {
“WebGUI::Group” :  [ “WebGUI::Workflow::Activity::EmailPoll”],
...
},
```

After you've made this change, restart WebGUI for the change to take effect.

Assets

An asset is the basic unit of content inside WebGUI. Assets can be files, images, articles, CSS, JavaScript, forums, calendars, galleries, applications, and more. Some assets are used to display other assets. Some assets are used to display and manage collateral data. Assets are the base of WebGUI, the main event, the big show, the way to get WebGUI to do what you need.

With the full power of the Session object and the rest of the WebGUI API, you can make assets that can do anything. With the SQL API, you can make an asset to manage your customer information. With the User and ProfileField API's, you can make new ways to display and manage user information.

The Bare Necessities: a Cliched Example

Making the simplest of assets is a simple process. To make it even easier, you could start with the asset skeleton, located in WebGUI/lib/WebGUI/Asset/_NewAsset.skeleton. The example here starts from scratch with the barest minimum. These lines set up your new asset as a subclass of WebGUI::Asset.

```
package WebGUI::Asset::HelloWorld;
use strict;
use base 'WebGUI::Asset';
```

Definition

The definition method gives information to your asset. The asset's properties, what table in the database, and the internationalized name of the asset are just part of the information you can get from the definition.

```
sub definition {
        my $class           = shift;
        my $session         = shift;
        my $definition      = shift;
```

These lines start your definition. sub definition is a class method, so the first argument will be the class. $session is a WebGUI::Session object, which you'll need later to get a WebGUI::International object (or other things). $definition is a definition from a subclass. Since definitions need to be inherited, each class's definition sub calls its parent's definition sub, and so on up the chain to WebGUI::Asset->definition(). The end result is an array reference of hash references, with your new class's information as the first element.

```
	tie my %properties, 'Tie::IxHash', (
```

Next, create a hash to store your asset's properties. In order to maintain the order of the hash, the example uses Tie::IxHash. This way, every time you call values %properties, or keys %properties, the information returned will be in the same order as they were added (read Perldoc Tie::IxHash for more information). To begin, add a property that will store what greeting you want to use.

```
	tie my %properties, 'Tie::IxHash', (
		greeting	=> {
			tab			=> 'properties',
			fieldType		=> 'text',
			defaultValue	=> 'Hello',
			label		=> 'Greeting',
			hoverHelp	=> 'What greeting to give the user',
		},
	);
```

These properties will show up inside of a WebGUI::TabForm, so asset properties are defined in terms of a WebGUI::Form::Control object with some additional properties added in. From top to bottom, you have:

tab

This is which tab of the WebGUI::TabForm this property will show up in. This will be discussed again later.

fieldType

The fieldType is the type of WebGUI::Form::Control this property is. This will prefix "WebGUI::Form::" to this value, and upper-case the first character, so this greeting is a WebGUI::Form::Text control.

defaultValue

This is the default value for this property. This is not only given to the WebGUI::Form::Control, but it is handled specially by the update() sub.

If you ever want this property to be "NULL" or undef, the defaultValue for this property must be undef.

label

This is the label that will be used for this property. This will usually be something from a WebGUI::International object, which is covered later.

hoverHelp

This will be shown when a user hovers over the label for this property. It is also something that should be part of a WebGUI::International object.

Additional keys in the property hash reference will be given directly to the form control object, so see the WebGUI::Form::Control subclass for more information. Properties are only one part of the definition, though they are the most important part.

```
    push @$definition, {
        properties          => \%properties,
        assetName               => 'Hello World',
        icon                    => 'assets.gif',
        tableName               => 'HelloWorld',
        className               => __PACKAGE__,
        autoGenerateForms       => 1,
    };
    return $class->SUPER::definition( $session, $definition );
}
```

Here's where you add your entry to this asset's definition and send it up the chain. From top to bottom:

properties

Here's where you give a reference to your hash of properties.

assetName

This is the name of the asset, which will show up on the edit form, the new content pane, and other places. This may be internationalized.

icon

This is a filename for an image to be used as this asset's icon. The asset icons are located in the WebGUI/www/extras/assets directory, with the smaller version located in WebGUI/www/extras/assets/smal/. One filename will be used in both folders.

tableName

This is the name of the table in the WebGUI database.

className

This is the name of the class you're making. Using the __PACKAGE__ special value makes this easy.

autoGenerateForms

If this is set to a true value, the asset edit form will be automatically generated from the properties. In most cases, you'll want to leave this set to 1. Advanced techniques with the asset edit form will be covered later.

The final line sends your definition to your superclass, which will in turn send it up until it reaches WebGUI::asset.

Though all of these keys are required, the most important is the tableName. This is the name of the table that this asset's properties will be stored in. Only this class's properties will be stored in this table, since each level of the definition has its own tableName. This is an important part of the asset system, and what makes subclassing easy.

Create the Database Table

Since your tableName is "HelloWorld", you must make a table in your database called "HelloWorld".

```
CREATE TABLE `HelloWorld` (
	`assetId` VARCHAR(22) BINARY NOT NULL,
        `revisionDate` BIGINT NOT NULL,
        `greeting` VARCHAR(255),
        PRIMARY KEY ( `assetId`, `revisionDate` )
);
```

Your table needs at least the assetId and revisionDate columns, with those exact properties. The assetId is a base64 string, so it needs the BINARY property in order to ensure that it's treated as case-sensitive. The revisionDate is an epoch time, so it needs a BIGINT so that it doesn't overflow a normal INT field.

Following the two required columns is your greeting column. This column has the same name as the key from the properties hash. Since it's a text field, assign a varchar type to it. Notice that you cannot tell this column to be NOT NULL, since new assets are initially inserted with only the assetId and revisionDate columns when they are added to the database. Only these columns can be NOT NULL.

The final line sets your primary key to be a combination of the assetId and revisionDate columns. Since assets are most often looked up by both assetId and revisionDate, this will significantly speed up the lookups.

View Method

The other required method is the view method. This method is the default view for the asset. It differs from www_view (and other www_ methods) in that this method will be called when the asset is part of a Page Layout. www_view is what is called when the asset is accessed directly, which in turn calls the view method. So for the default view, override the view method.

```
sub view {
                my $self = shift;
                my ($session, $user) = $self->session->quick( qw( session user ) );
                my $output   = join ', ', $self->get('greeting'), $user->username;
                return $output;
 }
```

The first two lines are self-explanatory. They set up the subroutine as an object method. The next line gives you a couple objects from the session, namely the session and user objects. The fourth line makes your output, using the get(property) method to get your greeting, and gets the username from the user object. Line five returns your output, which will then be printed to the user.

Add the Asset to the Configuration Table

In order to actually use a new asset in your site, you need to tell WebGUI about it. This is done through the configuration file.

There are three sections in the WebGUI configuration file that control which asset types are shown in the New Content menu of the admin accordion and the asset manager. The first one, “assets”, is used for general-purpose assets, and the Hello, World! asset would definitely fit in that category. The other two are used for more specialized asset types. “utilityAssets” is best used for those assets that are used less frequently by normal users. “assetContainers” is reserved for those assets that don't have any content of their own, but instead display other assets, like a Page Layout or a Folder.

To add your asset, change the “assets” configuration to look something like this:

```
“assets” : [
   “WebGUI::Asset::HelloWorld”,
   “WebGUI::Asset::Snippet”,
   “WebGUI::Asset::Redirect”,
   ...
],
```

Now, when you restart your server, your Hello World asset will be available in the New Content menu.

Hello World

The complete HelloWorld asset looks like this:

```
package WebGUI::Asset::HelloWorld;
use strict;
use base 'WebGUI::Asset';
sub definition {
        my $class           = shift;
        my $session= shift;
        my $definition       = shift;
        tie my %properties, 'Tie::IxHash', (
                greeting      => {
                        tab                     => 'properties',
                        fieldType               => 'text',
                        defaultValue=> 'Hello',
                        label                   => 'Greeting',
                        hoverHelp               => 'What greeting to give the user',
                },
        );
        push @$definition, {
                properties                      => \%properties,
                assetName                       => 'Hello World',
                icon                            => 'assets.gif',
                tableName                       => 'HelloWorld',
                className                       => __PACKAGE__,
                autoGenerateForms                       => 1,
        };
        return $class->SUPER::definition( $session, $definition );
}
sub view {
        my $self = shift;
        my ($session, $user)
                = $self->session->quick( qw( session user ) );
        my $output
                = join ', ', $self->get('greeting'), $user->username;
        return $output;
}
1; # The truth is right here
```

Adding a Template

Time for a more fun example: make an InfoCard to store some personal information about a person. While you're at it, add a template so that users of your asset can decide how to display that information. Start out with making your new properties. First, make a template.

```
 sub definition {
                my $class             = shift;
                my $session= shift;
                my $definition       = shift;

                tie my %properties, 'Tie::IxHash', (
                    templateIdView => {
                            tab                     => 'display',
                            fieldType               => 'template',
                            label                   => 'Template',
                            hoverHelp               => 'Template for this InfoCard',
                            namespace               => 'InfoCard',
            },
```

The templateIdView property will use WebGUI::Form::Template (line 9) to make a select box populated with templates in the InfoCard namespace (line 12). This property will show up on the display tab (line 8), which is the standard place for template select boxes.

For your additional properties, make some fields for the person's name, address, city, state, zip, country, phone, and e-mail.

```
                    name => {
                    tab                     => 'properties',
                            fieldType               => 'text',
                    label                   => 'Name',
                    hoverHelp               => 'The persons name',
                    },
                    address => {
                    tab                     => 'properties',
                    fieldType               => 'textArea',
                    label                   => 'Address',
                    },
                    city => {
                            tab                     => 'properties',
                            fieldType               => 'text',
                            label                   => 'City',
                    },
                    state => {
                    tab                     => 'properties',
                            fieldType               => 'text',
                    label                   => 'State',
                    size                    => 2,
                    },
                    zip => {
                    tab                     => 'properties',
                            fieldType               => 'zipcode',
                    label                   => 'Zip Code',
                    },
                    country => {
                    tab                     => 'properties',
                            fieldType               => 'country',
                    label                   => 'Country',
                    },
                    phone => {
                    tab                     => 'properties',
                            fieldType               => 'phone',
                    label                   => 'Phone',
                    },
                    email => {
                    tab                     => 'properties',
                            fieldType               => 'email',
                    label                   => 'E-Mail',
                    },
            );
```

Finally, to round out your definition method, you'll need your other properties.

```
                push @$definition, {
                        properties              => \%properties,
                        assetName               => 'InfoCard',
                        icon                    => 'assets.gif',
                        tableName               => 'InfoCard',
                        className               => __PACKAGE__,
                        autoGenerateForms       => 1,
                };
        }
```

The creation of the InfoCard table is left as an exercise for the reader.

getTemplateVars

Once you have your definition, you can create your view. This time, however, instead of just returning a bunch of HTML, you're going to build a template. The first thing you need to do is build a method to get the variables you're going to give the template processor. Call your method getTemplateVars.

```
	sub getTemplateVars {
		my $self		= shift;
		my $var		= $self->get;

		$var->{ url } = $self->getUrl;

		return $var;
	}
```

Since most of the interesting data is part of your asset's properties, start out with that. The get method, when given no arguments, returns a hash reference of the asset's properties. Since the asset's URL property may not be the same URL that the user needs to access your asset, due to gateway or other considerations, override the URL property with the value from getUrl, which gets an absolute URL to your asset (including gateway, but not including the domain). Finish up by returning this hash reference.

There are other things that look for a getTemplateVars method before defaulting to the get method. It is good practice to have a getTemplateVars method. getTemplateVars should return the template vars needed for every view of the asset, including those that may be useful outside the asset's view methods.

processTemplate

Once you have your template vars, you need to put them inside a template. Since this is such a common operation, there's an easy way to do it: processTemplate.

```
sub view {
        my $self                = shift;
        my $var                 = $self->getTemplateVars;
        return $self->processTemplate( $var, $self->get('templateIdView') );
}
```

processTemplate will instantiate the WebGUI::Asset::Template object with the given ID (the second argument), and give the processor the hash reference of variables (the first argument). It then returns the value from the template's process method. The end result is the template gets processed and you get your output.

prepareView

One of WebGUI's many features is Content Chunking. In short, this means that the user is given something, anything, before the real process intensive operation (the view method probably) is run. In order to make this happen, you need to be able to send the HTTP header and the site header (which includes the <head> block) before you run the view method, but your asset (and its template) may have tags it wants to add to the <head> block.

For this reason, there's the prepareView method. prepareView prepares the template that is to be used in the view method.

```
sub prepareView {
        my $self                = shift;
        $self->SUPER::prepareView();
        my $templateId      = $self->get('templateIdView');
        my $template
                = WebGUI::Asset::Template->new( $self->session, $templateId );
        $template->prepare;
        $self->{_viewTemplate} = $template;
        return;
}
```

Starting with line 3, call the superclass prepareView. The top-level, WebGUI::Asset::prepareView prepares the head tags for the asset and adds them to the queue to be printed. Lines 4-7 instantiates your template and prepares it, which adds its own head tags to the queue to be printed. Line 8 adds your template to the object, which you'll need later, and line 9 is the best practices method of ending a sub with nothing returned.

Now, you need to alter your view method to use the prepared template.

```
        return $self->processTemplate( $var, undef, $self->{_viewTemplate} );
```

The third argument to processTemplate allows you to pass in the already-prepared template.

The completed WebGUI::Asset::InfoCard:

```
package WebGUI::Asset::InfoCard;
use strict;
use base 'WebGUI::Asset';
use Tie::IxHash;
sub definition {
    my $class       = shift;
    my $session= shift;
    my $definition      = shift;
    tie my %properties, 'Tie::IxHash', (
        templateIdView => {
            tab             => 'display',
            fieldType       => 'template',
            label           => 'Template',
            hoverHelp       => 'Template for this InfoCard',
            namespace       => 'InfoCard',
        },
        name => {
            tab             => 'properties',
            fieldType       => 'text',
            label           => 'Name',
            hoverHelp       => 'If you need help here, try a professional.',
        },
        address => {
            tab             => 'properties',
            fieldType       => 'textArea',
            label           => 'Address',
        },
        city => {
            tab             => 'properties',
            fieldType       => 'text',
            label           => 'City',
        },
        state => {
            tab             => 'properties',
            fieldType           => 'text',
            label               => 'State',
            size                => 2,
        },
        zip => {
            tab                 => 'properties',
            fieldType           => 'zipcode',
            label               => 'Zip Code',
        },
        country => {
            tab                 => 'properties',
            fieldType           => 'country',
            label               => 'Country',
        },
        phone => {
            tab                 => 'properties',
            fieldType           => 'phone',
            label               => 'Phone',
        },
        email => {
            tab                 => 'properties',
            fieldType           => 'email',
            label               => 'E-Mail',
        },
    );
    push @$definition, {
        properties              => \%properties,
        assetName               => 'InfoCard',
        icon                    => 'assets.gif',
        tableName               => 'InfoCard',
        className               => __PACKAGE__,
        autoGenerateForms       => 1,
    };
}
sub getTemplateVars {
    my $self            = shift;
    my $var             = $self->get;
    $var->{ url    } = $self->getUrl;
    return $var;
}
sub prepareView {
    my $self            = shift;
    $self->SUPER::prepareView();
    my $templateId      = $self->get('templateIdView');
    my $template
        = WebGUI::Asset::Template->new( $self->session, $templateId );
    $template->prepare;
    $self->{_viewTemplate} = $template;
    return;
}
sub view {
    my $self            = shift;
    my $var             = $self->getTemplateVars;
    return $self->processTemplate( $var, undef, $self->{_viewTemplate} );
}
1;
```

The Edit Form

You now have an asset that stores a person's information and displays that information using a template. Now the powers that be have decided that you need to keep track of updates to a person's information, what was updated, and, most importantly, why. Luckily for you, with the asset versioning system, WebGUI will already keep track of what was updated, but you'll need to keep track of the why.

getEditTabs

Instead of putting this on the properties tab with the rest of the information, this example makes a new tab called "History". To add more tabs to the asset edit form, override the getEditTabs method.

```
        sub getEditTabs {
                my $self            = shift;
                return (
                        [ 'history', "History" ],
                );
        }
```

getEditTabs returns a list of array references. The first element in the arrayref is the ID of the tab, which you'll use in the definition sub. The second element is the label, which is the friendly label for the user. Now, if you go to your asset edit page, you'll see an additional tab called History, but you've yet to put any properties in it. Add a text area to the history tab by adding it to the properties in the definition sub.

```
        history => {
        tab                 => 'history',
            fieldType           => 'textArea',
        label               => 'History',
        },
```

Once you add history to your data table, you'll be able to edit the history from your asset edit page, but now you'll also be able to edit the old history, which shouldn't be allowable. You're going to have to do something different. Add the old history as text on the history tab, and have the form element be for adding to the history (instead of changing the history).

getEditForm

The getEditForm method returns the WebGUI::TabForm object that will render the asset edit form. If you want to change how the edit form looks, you can override it and add to it as you wish. If you want to replace the elements you defined in the definition properties, first you need to tell WebGUI not to add the element automatically with the autoGenerate key:

```
        history => {
            tab                 => 'history',
            fieldType           => 'textArea',
            label               => 'History',
            autoGenerate        => 0,
        },
```

By setting autoGenerate to some false value (0, undef, "", or "0"), the default getEditForm method will not add the property field to the TabForm, so you can add it yourself by overriding the getEditForm method.

```
    sub getEditForm {
        my $self            = shift;
        my $tabform         = $self->SUPER::getEditForm;
        $tabform->getTab('history')->raw('<pre>' . $self->get('history') . '</pre>');
        $tabform->getTab('history')->textArea({
            name        => 'history',
            value       => ( $self->session->form->get('history') ),
        });
        return $tabform;
    }
```

Start by getting the initial tabform by calling the superclass method. Next, add the old history as plain text to the history tab. The getTab method takes a single argument, the ID of the tab, and returns a WebGUI::HTMLForm object. Then, call raw on that object to put in some raw HTML.

After you add the old history, add the form element for the new history as shown in lines 5-8. Give it the same name as the property from the definition, and a value from the current form submit. Get the value from the session in case the form gets shown again after it's submitted, which can happen if there are any errors in the form.

Finally, return the tabform object so that it can be rendered. Now, your form looks like you want it to, but you're still resetting the history every time you hit the Save button. How can you take the value of the history element and add it to the old history?

processPropertiesFromFormPost

Pressing the Save button on the asset edit form runs the www_editSave method. This method has the appropriate magic for adding and editing assets, but the method that actually saves the information from the edit form is processPropertiesFromFormPost. By default, it overwrites all the properties it gets, which will destroy your old history, so you're going to have to tell it not to by using the noFormPost key in the definition.

```
        history => {
                tab             => 'history',
                fieldType       => 'textArea',
                label           => 'History',
                autoGenerate    => 0,
                noFormPost      => 1,
        },
```

Now the old value won't get clobbered by the new value, but you still haven't done anything to add the new value. Do so by overriding the processPropertiesFromFormPost method.

```
    sub processPropertiesFromFormPost {
        my $self            = shift;
        my $form            = $self->session->form;
        my $errors          = $self->SUPER::processPropertiesFromFormPost || [];
```

First, call the superclass processPropertiesFromFormPost. Since it may return an array reference of errors, try to grab it. If it doesn't, initialize your errors with an empty array reference. You need $errors to be an array reference for the remainder of the sub.

```
        unless ( $form->get('history') ) {
            push @$errors, "You must update the History!";
        }
        return $errors if @$errors;
```

Next, verify that the user entered some history text. If the user didn't, add the error to your list of errors. After compiling your list of errors, you can return the array reference if necessary.

```
        # Data passes all checks, save
        $self->update({
            history => join "\n", $form->get('history'), $self->get('history'),
        });
        return;
    }
```

After you've verified your data, you can update your asset. Add your new history on top of your old history. When you save your asset, you're required to add a reason why you're editing it, and that reason is saved along with all the old reasons.

www_edit

The final problem from the powers that be is that the asset edit form is part of the Admin Console. Fortunately, this is easy enough to fix, as the most important part of the www_edit page is handled by getEditForm. The only important things that www_edit does is check that the user has privileges, check that the asset is unlocked, and return the edit form.

```
    sub www_edit {
        my $self            = shift;
        return $self->session->privilege->noAccess unless $self->canEdit;
        return $self->session->privilege->locked unless $self->canEditIfLocked;
        return $self->getAdminConsole->render( $self->getEditForm->print );
    }
```

This is the default www_edit sub. To remove the asset from the Admin Console, you just need to add this subroutine to your asset and edit line 5 like so:

```
		return $self->getEditForm->print;
```

Your edit form will now appear outside of the Admin Console. Your new code looks like this:

```
sub getEditForm {
	my $self		= shift;
	my $tabform			= $self->SUPER::getEditForm;
	$tabform->getTab('history')->raw('<pre>' . $self->get('history') . '</pre>');
	$tabform->getTab('history')->textArea({
		name		=> 'history',
		value		=> ( $self->session->form->get('history') ),
	});
	return $tabform;
}
sub getEditTabs {
	my $self		= shift;
	return (
		[ 'history', "History" ],
	);
}
sub processPropertiesFromFormPost {
	my $self		= shift;
	my $form		= $self->session->form;
	my $errors		= $self->SUPER::processPropertiesFromFormPost || [];
	unless ( $form->get('history') ) {
		push @$errors, "You must update the History!";
	}
	return $errors if @$errors;
	# Data passes all checks, save
	$self->update({
		history			=> join "\n", $form->get('history'), $self->get('history'),
	});
	return;
}
sub www_edit {
	my $self		= shift;
	return $self->session->privilege->noAccess unless $self->canEdit;
```

```
        return $self->session->privilege->locked unless $self->canEditIfLocked;
        return $self->getEditForm->print;
}
```

More Definition Magic: Filters

By using another key in the definition, you can automatically apply filters to all data in your asset properties. Say you're thinking ahead and want to format the address property to be HTML-ized, based on the plain text input. You don't just want to format it from the asset edit form. You also want anyone adding your asset programmatically to have to go through this filter too.

Instead of using processPropertiesFromFormPost(), which would only cover users adding your asset from the asset edit form, you can use the “filter” key in the properties part of your definition. This defines a subroutine in your asset class to use to filter the data for that property.

```
    address => {
        tab             => 'properties',
        fieldType   => 'textArea',
        label       => 'Address',
        filter      => 'filterAddress',
    },
```

Here, define a filter called “filterAddress”. You need a method to handle the filtering. This method takes the string to be filtered as the first parameter, and returns the value to be set in the database.

```
sub filterAddress {
    my $self     = shift;
    my $string  = shift;
    my $output  = $string;

    # First change newlines into HTML
    $output    =~ s{\n}{<br />}g;

    # Wrap the whole thing in address tags if necessary
    if ( $output !~ s{^<address>} ) {
        $output     = '<address>' . $output . '</address>';
    }

    return $output;
}
```

Now, whenever the address is changed, your filterAddress method will be run to make sure that it adheres to the appropriate format.

Tips, Tricks, and Things Left Unsaid

- When something is wrong, step one is always to check the error log.

  If you visit your asset and you see a blank screen, check the WebGUI error log. If you don't see what you expect to see, check the WebGUI error log. If things are bad enough that you see Apache's Internal Server Error page, then check Apache's error log.

  If the WebGUI error log doesn't have anything, make sure you bump up the log level from ERROR to INFO or DEBUG. See etc/log.conf for more details.

- When using WebGUI::Session::Privilege to return an error message, use noAccess() over insufficient(). noAccess gives a login form for those who are not logged in, and after they log in, they are taken right to where they wanted to be instead of to the default view of the asset. See WebGUI::Session::Privilege for more information.

- The default processTemplate() method adds a bunch of things to the variables passed into it, including the asset's metadata, all the asset's properties, and the controls for use in admin mode. The controls should be added to each template you create. It's located in <tmpl_var controls>. Use <tmpl_if session.var.adminOn> to show only the controls when admin mode is on.

- By default, the canAdd() method checks if the user is allowed to add any asset of the given class. It does so by checking the config file, and falls back to the Turn Admin On group. As a rule, canAdd() should check the lowest possible level of privileges: if anyone could be allowed to add your asset, canAdd should just return true. If you receive an error when trying to add your new asset, check canAdd first.

- When saving a new asset, the user also has to pass a canEdit() check on the parent asset. If you're writing an asset that will contain other assets, you will probably need to make a special case in canEdit() to handle adding new assets. If you checked canAdd and are still receiving errors trying to add your new asset, check here next. Note that the canAdd check is done before you see the form, and the canEdit check is done after you fill in the form and click Save.

- You can prevent certain classes from being added underneath your asset by overriding addChild() and returning undef if the asset being added doesn't pass your checks. This should only be done to make sure the wrong assets don't get added as children of your asset, since there will be no useful error message shown to the user.

Lineage

Every asset in WebGUI is part of the asset tree. Like a directory tree, the asset tree starts with a Root asset. Every asset descends from the Root asset. And, like a directory tree, every asset (except the Root asset) has a parent. In WebGUI, many assets can contain other assets. Some assets can only contain specific assets, like Calendars can only contain Events. The relationship between a parent asset and its child is stored in the asset table as "parentId".

Using the parentId, you can calculate the ancestry of any asset in the tree by walking up and down the tree, using parentIds to determine how to navigate, but this is unbearably slow. You'd need to query the database for every level. To solve this problem, WebGUI uses lineage.

The lineage of an asset is like a cache of the asset's position in the tree. Lineage is stored as a sequence of digits in the lineage column of the asset table, with 6 digits per level. Say you have a Page Layout, inside a Folder, with two Articles (Article1 and Article2), that each have two Images (Image1a, Image1b, Image2a, and Image2b). Using lineage, you can quickly look at the assets that are related to each other.

assetID	**Lineage**
Root	000001
Page	000001000001
Article1	000001000001000001
Image1a	000001000001000001000001
Image1b	000001000001000001000002
Article2	000001000001000002
Image2a	000001000001000002000001
Image2b	000001000001000002000002

Since every asset is descended from the root asset, every asset's lineage starts with the same six digits "000001". The Page Layout asset is a child of the Root asset, so it has an additional six digits. Every new child that is added will get a unique 6-digit number added to its parent's lineage, so the Article1 asset, a child of the Page asset, has another six digits in its lineage. The same with the Image1a asset.

Looking closer at the Image1a asset's lineage, you can determine a few things:

Ancestors	000001 000001	Page
Parent	000001 000001 000001	Article1
*Self	000001 000001 000001 000001	Image1a
Siblings	000001 000001 000001 ------	Siblings have the same parent
Children	000001 000001 000001 000001 ------	Children have Image1a as a parent.
Descendants	000001 000001 000001 000001 ++++++	Descendants have Image1a in their ancestry.

If you know an asset's lineage, you can instantiate it using the newByLineage() method of the WebGUI::Asset class (located in the AssetLineage mix-in). However, for most purposes you're going to want "all children of this asset", or "all descendants of this asset." For this, you can use the getLineage() method (also in the AssetLineage mix-in).

The getLineage Method

getLineage is the standard way to get particular assets from the asset tree. As such, it is an extremely powerful and flexible tool. At its simplest, it gets all the assets with a certain relationship to the current asset. At its most complex, it can select only particular types of assets, at particular positions, and even return instantiated objects for you.

```
        getLineage( relations [, options ] )
```

“relations” is an array reference of relations to get. It can consist of the following values:

self	Add the current asset to the list of returned.
siblings	Add siblings of the current asset to the list returned.
children	Add children of the current asset to the list returned.
descendants	Add any descendant (including children).
ancestors	Add the ancestors.

So, if you want to get all of the children of the Article1 asset from above:

```
        $article1->getLineage( [ 'children' ] );
        > [ 'Image1a', 'Image1b' ]
```

Or, if you want to get every ancestor all the way up to the root asset:

```
        $article1->getLineage( [ 'ancestors' ] );
        > [ 'Root', 'Page' ]
```

If you want to get every descendant of the Page asset, and include the Page asset itself:

```
        $page->getLineage( [ 'self', 'descendants' ] );
        > [ 'Page', 'Article1', 'Image1a', 'Image1b', 'Article2', 'Image2a',        'Image2b' ]
```

Query the Asset Tree

What if you only want to get certain types of assets in your lineage? Or only a certain number? For this, you have “options.” options is a hash reference with various keys that control the type and number of assets you get.

In order to control which types of assets you get, you can use the includeOnlyClasses and excludeClasses options. These options allow you to specify an array reference of class names to include or exclude from your results, like so:

```
        $page->getLineage( [ 'descendants' ], {
                includeOnlyClasses => [ 'WebGUI::Asset::File::Image' ]
        }
        > [ 'Image1a', 'Image1b', 'Image2a', 'Image2b' ]

        $page->getLineage( [ 'descendants', {
                excludeClasses => ['WebGUI::Asset::File::Image']
        }
        > [ 'Article1', 'Article2' ]
```

By default, getLineage will only get assets that have a state of “published” and will do the right thing when it comes to versioning (status). For most purposes, this is what you want, since assets in the trash, in the clipboard, or in an open/pending version tag that you're not part of should not be part of your normal operations. However, for the times when you need assets with special states or statuses, you have the statesToInclude and statusToInclude options:

```
        # Get all children of $asset that are pending
        my $pendingChildren
                = $asset->getLineage( [ 'children' ], {
                        statusToInclude => [ 'pending' ],
                } );

        # Get all assets that are in the trash. $rootAsset is the Root asset
        my $assetsInTrash
                = $rootAsset->getLineage( [ 'descendants' ], {
                        statesToInclude => [ 'trash' ],
                } );
```

Since getLineage is the standard way to get assets out of the asset tree, there are ways to further limit which assets you get from the database.

The 'whereClause' option allows you to add an additional SQL where clause to the query. By default, all of the columns in the asset and the assetData table are available, so you can do things like...

```
        # Get all Post and Thread assets that were created in the last 24 hours
        # (Thread is a child asset of Post)
```

```
	my $todaysPostsAndThreads
		= $rootAsset->getLineage( [ 'descendants' ], {
			includeOnlyClasses => [ 'WebGUI::Asset::Post',
						'WebGUI::Asset::Post::Thread' ],
			whereClause
				=> 'creationDate >
					UNIX_TIMESTAMP( DATE_SUB(NOW(), INTERVAL 1 DAY) )
',
		} );

	# Get any asset that was added or modified in the last week
	my $whatsNew
		= $rootAsset->getLineage( [ 'descendants' ], {
			whereClause
				=> 'revisionDate >
					UNIX_TIMESTAMP( DATE_SUB(NOW(), INTERVAL 7 DAY) )
',
		} );

	# Get Wiki pages that were last modified by a certain user
	my $wikiActivity
		= $rootAsset->getLineage( [ 'descendants' ], {
			whereClause		=> 'revisedBy = "3"',
		} );
```

The 'joinClass' option allows you to specify an asset class and join all the tables that it uses so you can run queries against it. Combined with the 'whereClause' option, you can get very specific about the assets you get.

```
	# Get all Event assets that occur on December 25, 2008
	my $christmasEvents
		= $rootAsset->getLineage( [ 'descendants' ], {
			joinClass		=> 'WebGUI::Asset::Event',
			whereClause			=> 'Event.startDate = "2008-12-25"',
		} );
```

Due to the nature of inheritance and the class hierarchy, joinClass has a useful side-effect: joining a class will also include all child classes in the results (though you can't search on their attributes). If you don't want this side-effect, you can use the includeOnlyClasses or excludeClasses options.

```
# Get all Image and Photo assets that are JPGs
my $jpegs
        = $rootAsset->getLineage( [ 'descendants' ], {
                joinClass               => 'WebGUI::Asset::File::Image',
                whereClause                     => 'filename REGEXP "jpe?g"',
        } );

# Only get Image assets, not Photo assets.
my $jpegs
        = $rootAsset->getLineage( [ 'descendants' ], {
                joinClass                       => 'WebGUI::Asset::File::Image',
                includeOnlyClasses      => [ 'WebGUI::Asset::File::Image' ],
                whereClause                             => 'filename REGEXP "jpe?g"',
        } );
```

As you get even more specific with your queries of the asset tree, it becomes important to be able to modify the order and number of the assets you get. By default, the assets are sorted by their "lineage" field, which is also known as sorting by rank (more on rank later). Using the orderByClause option, you can order by any field to which you have access.

```
# Get the most recent content in descending order
my $whatsNew
        = $rootAsset->getLineage( [ 'descendants' ], {
                orderByClause           => 'revisionDate DESC',
        } );

# Get the threads in the message board $mboard ordered by their rating
my $recentThreads
        = $mboard->getLineage( [ 'descendants' ], {
                joinClass                       => 'WebGUI::Asset::Post::Thread',
                includeOnlyClasses      => [ 'WebGUI::Asset::Post::Thread' ],
                orderByClause           => 'threadRating DESC',
        } );
```

The above examples will return the assets in the order you want, but it will also return all the assets that match the query. Often you don't want all the assets, but only a few or only from certain depths in the tree. You have three ways to limit the number of assets you get. ancestorLimit and endingLineageLength establish a window of tree depths to retrieve, and limit determines the number of assets you get.

```
        # Only get three levels down from the current asset $asset
        my $threes
                = $asset->getLineage( [ 'descendants' ], {
                        endingLineageLength     => $asset->getLineageLength + 3,
                } );

        # Only get three levels up from the current asset $asset
        my $christmasList
                = $asset->getLineage( [ 'ancestors' ], {
                        ancestorLimit                   => 3,
                } );

        # Get the first Page Layout asset in our ancestry
        my $parentPage
                = $asset->getLineage( [ 'ancestors' ], {
                        includeOnlyClasses      => [ 'WebGUI::Asset::Wobject::Layout' ],
                        limit                   => 1,
                } );
```

To pull it all together, here are a few more real-world examples of uses of getLineage.

```
        # Get the next five upcoming events from the calendar $calendar
        my $comingUp
                = $calendar->getLineage( [ 'children' ], {
                        joinClass               => 'WebGUI::Asset::Event',
                        whereClause
                                => q{startDate >= CURDATE() and
                                        (startTime IS NULL or startTime >= CURTIME())},
                        orderByClause   => 'startDate, startTime',
                        limit                   => 5,
                } );

        # Get all the Threads started by the user with userId of '3' ordered by age
        my $adminThreads
                = $rootAsset->getLineage( [ 'descendants' ], {
                        includeOnlyClasses      => [ 'WebGUI::Asset::Post::Thread' ],
                        whereClause                     => 'ownerUserId = "3"',
                        orderByClause           => 'creationDate DESC',
                } );
```

addChild

Just as getLineage is the standard way to get assets from the asset tree, addChild is the standard way to put assets into the asset tree. addChild will instantiate the new asset, add it to the database, set the asset's parent to the current asset, and give you the new asset object, like so:

```
my $image2c = $article2->addChild( $properties );
```

Using more arguments to addChild, you can specify the assetId of the new asset, and the revisionDate. This is useful when creating asset installers, or other times when you decide on an assetId ahead of time, and must make sure that assetId is used.

```
# Create a new asset with a specific assetId that is shown as
# being created 60 seconds ago
my $image2d = $article2->addChild( $properties, 'Image2d', time - 60 );
```

Finally, the fourth argument is a hash reference of options. Currently, only a single option exists, skipAutoCommitWorkflows, but it is very important when writing new code that requires use of addChild. Without this option, if the asset you're adding has an auto-commit workflow, a new version tag will be created. Worse, since the version tag isn't sent through the commit process until requestAutoCommit is called, the version tag remains open, showing up in the Version Tags for anyone to join.

For most cases, when using addChild in your own application, you're going to need to use skipAutoCommitWorkflows.

```
# Add a new post but keep the same working tag
my $post
    = $thread->addChild( $properties, undef, undef,
        { skipAutoCommitWorkflows => 1 }
    );
```

Notice that you set both the assetId and revisionDate to "undef". This will cause addChild to generate them automatically, like you want.

Rank and Depth

When an asset is created, it's given a parentId, which is the assetId of the asset's parent, and a lineage, which is the same lineage as the asset's parent plus an additional six digits. Those six digits are also known as the asset's rank. As mentioned above, by default, getLineage sorts the assets it returns by rank, so rank can be important in making sure asset one is shown before asset two.

To modify an asset's rank, you can use the demote, promote and, if necessary, the swapRank methods.

```
# Move an asset above it's next-highest-ranked sibling
$asset->promote;

# Move an asset below it's next-lowest-ranked sibling
$asset->demote;

# Swap two assets in the rank
$asset1->swapRank( $asset2->get('lineage') );
```

These methods move an asset around on the same level of depth, but sometimes you want to move the asset to another place entirely. To do so, use the setParent method.

```
# Move $asset to a new parent $parentAsset
$asset->setParent( $parentAsset );
```

In addition to updating the parentId of $asset, setParent also calls cascadeLineage, which updates the lineage of $asset and all of the descendants of $asset. This is very important, so be sure to use setParent when you want to move assets up and down the asset tree.

Writing Wobjects

Wobject is short for web object and serves as the plugin type for creating custom applications for your site in WebGUI. Because wobjects are both pluggable and true objects, the possibilities of what you can do are nearly limitless. Applications as simple as a guest book and as complex as complete Human Resources Information Systems have been created for WebGUI. The only limiting factor when it comes to what you can accomplish with wobjects is you.

History

WebGUI has had a pluggable application interface since its inception in 2001. To better understand wobjects and what they are, it's helpful to take a look back at how wobjects evolved into what they are today.

Up until version 5 of WebGUI, Wobjects were called Widgets. If you have been using WebGUI since before version 5, it's possible that you might still have the widget table in your database. To create a new widget for WebGUI, you basically needed to create a new WebGUI::Widget package, add a www_view and www_edit method, and then add the widget to WebGUI's config file, similar to what is still done today.

While they were extremely simple to write (no extended API's existed way back in the beginning), there was a lot of necessary repetition. Each time you created a widget you had to basically re-write the edit screen. If you wanted to add a feature to an existing widget, you had to copy it and all of its database tables before you could do so. There was nothing you could do to control the style. Every widget, including the edit page, was wrapped in the style of the site.

Widgets were not flexible enough to accommodate advanced applications, and their repetitiveness made the development cycle long and tedious with a good deal of code copying. WebGUI 5 introduced the wobject, which replaced widgets and introduced an API for developing applications within WebGUI. Wobjects solved the problem of repetition by including some convenience methods for adding the edit form and handling data; however, it was still limiting in that Wobjects in WebGUI 5 weren't true objects, so you still needed to fully copy wobjects to make changes. Style handlers were added which allowed some flexibility when it came to styles; however, wobjects were still limited in that they had to have a style. You couldn't return raw data, such as RSS feeds.

WebGUI 6 introduced the modern wobject as you know them today. Wobjects became true objects which allowed for inheritance. Style became completely flexible, allowing raw data to be returned by an application, and the API was enhanced to further limit the development cycle by introducing auto generating forms based off a definition. Wobjects became truly powerful, capable of performing any task regardless of how complex.

What is a Wobject?

Now that you have a little history about the evolution of wobjects, let's talk a bit more about what the modern wobject is and what it provides to you, the developer. Wobjects in WebGUI are simply assets. A full chapter in this book is dedicated to developing assets. If you have read and understand that chapter then there is relatively nothing new to learn to develop a wobject.

So what's the difference then? Why have a whole chapter on writing wobjects if they are simply assets?

The main difference between assets and wobjects is that wobjects give you API with convenience methods for handling collateral data and have a built in style wrapper. What this means is that you don't have to manually look up the current style of the page and wrap your asset inside it. This is done automatically for you. So how do you know whether to write an “Asset” or a “Wobject”? Ask yourself a few questions:

1. Does your asset take on its own “focus”?

   Let's digress for a moment and discuss a concept called “Wobject focus” in WebGUI. Start by imagining a discussion forum on a page of your WebGUI website called “Discussion” with URL /discussion. When you go to the discussion page, you see a form within the current style of your site (you likely have a top banner, maybe left hand navigation, and a footer) along with any other assets you may have dropped on to this page. For the sake of argument, let's say you have a second discussion forum on this page.

   Now, imagine you want to view all of the threads within the “General” topic of the first discussion forum. This is going to take you to a new url: /discussion/forum1?func=viewThreads;threadId=xxxx. This is the URL of the wobject. The page which contained both discussion forums loses focus and the wobject, which displays the threads within the discussion forum you clicked, gains focus. What this means is that everything else on the page (the second discussion forum) is no longer displayed. Only the wobject you've clicked on is displayed. This is most visible when you return to main view of the wobject. /discussion/forum1?func=view. This will return you to the main view of the forum; however, the second discussion forum will not show up on the page. This is because the first discussion forum still has focus. To see them both, you need to go back to the page where both of the discussion forums reside, /discussion.

If your wobject gains focus, you will need to wrap it in a style so that it looks like the rest of the site. Otherwise, users will be confused when your wobject gains focus as it will look completely different than the page they were just on. As stated before, wobjects have a built in style wrapper which automatically inherits from the parent when users add it to the site.

If you anticipate situations in which your wobject will need to gain "focus", you probably want to create a wobject instead of an asset.

2. Is there collateral data associated with your asset?

   Let's say you want to build a recipe book asset which allows users to enter and search for their favorite recipes. As you learned in the chapter on writing assets, each asset needs to store metadata associated with it so WebGUI knows how to display various aspects of it. You would start building this asset much in the same way by creating a table that had the assetId and revisionDate fields. This asset, however, needs to have multiple recipes associated with each instance. In order to do that you are going to need at least one more database table that stores information about each recipe. This is what is called collateral data.

   If you anticipate needing collateral data, you probably want to create a wobject instead of an asset.

3. Are there multiple UI components of your asset?

   In the recipe asset example above, you are going to need more than just the asset add/edit and view screens. You are going to need a screen which allows users to add, edit, view, and remove recipes from the system. These are separate UI components which you will expose to your users, and therefore require their own templates.

If you anticipate needing multiple UI components, you probably want to create a wobject instead of an asset.

Getting Started

Since you already know how to develop assets from reading the Assets chapter, this discussion will dive into a little more of the functional and technical planning that is necessary when developing a Wobject. For all intents and purposes, a wobject is a full featured application within WebGUI which requires planning before you dive into the code. The planning stage is divided into two parts: the Functional Specification and the Technical Specification.

Writing a Functional Specification

One of the most important, if not *the* most important part of designing an application (especially an application for the web) is to try to anticipate how your users are going to attempt to use it. Users have some preconceived notions of how software works. This is what is called the “User Model,” and it is what you should attempt to identify before writing any piece of software. The document that describes your vision of the “User Model” is called a Functional Specification for the software.

Writing a specification is a lot like flossing: you know you should be doing it, but you never do. Most people will tell you they don't write specifications because it saves time. That may be true initially, but a bigger problem is that most developers get an idea and want to run with it before organizing all of their thoughts.

Let's clear one thing up before going any further. Writing a functional specification may not be the most enjoyable thing to do, but for any project that takes longer than a week to code, not writing one will almost certainly lead to an even less enjoyable practice. The practice of patching up software that is full of holes.

A functional specification describes how your application will work entirely from the user's perspective. It doesn't care how it is implemented. It talks about features and it shows screen shots of every page trying to visually describe the application and the user's experience. This doesn't need to be a long document, but it should be long enough to fully express your idea.

You also don't need to create flashy graphics. In fact, simply drawing out the screen on a piece of paper and scanning it would suffice. If you are so inclined, applications like Microsoft Visio (Windows)® or Omnigraffle (Mac)® have tools that let you lay out simple forms. Defining the flow of the user's experience is what is important.

A functional specification should contain the following:

- Description: a brief description of the application, no longer than one or two small paragraphs. Include the goal of the project and a high level overview of what the software does.
- User Roles: bullet the various user roles (admins, managers, approvers) that exist in the application, describing each briefly in one or two sentences.
- Rules: bullet any rules that must be followed at all times by the software. For example, "An administrator must approve a document before it's published to the site."
- External Processes: briefly describe all interactions with any third party software. For example, "Data is retrieved from XYZ RSS feed on yahoo.com."
- Future Considerations: detail all functionality that will NOT be included in this release of the application, but may be considered in the future. This will force you to think about possible stubs you may want to add for future development.
- Workflow Diagram: provide a workflow diagram of how data moves through the system.
- Screen Specifications: provide a screen shot of each screen of the application as you envision it, and briefly describe what actions can occur. This will not only make it easier to figure out how your application should be implemented, but it ensures that you don't leave anything out that you might need.

Writing a Technical Specification

The technical specification describes the internal implementation of the application. It talks about data structures, relational database models, algorithms, etc. It sets the parameters for building this piece of software.

A Technical Specification should include:

- Entity Relationship Diagram (ERD): provide a graphical representation of the different database tables and their relationships. This will help you identify that various components that need to be created.
- Flow Charts: provide a diagram that shows the different components of the application and how they fit together. This will allow you to see a visual representation of your application, making it easier to find any holes that may need to be filled.
- Method Library: describe any methods you need to create along with their signatures. This helps you better visualize the entire application and what needs to be done.
- High Level Pseudocode: for complex ideas, it may be a good idea to provide some pseudocode that describes how you anticipate things working. This is useful as it allows you to easily share your ideas with others and identify shortcomings and pitfalls.

Building Your Wobject

Before beginning, take a look at the following Hello World example. This illustrates building a wobject at its most basic level, and is a helpful reference before moving on to a more advanced, step by step, example.

Hello World

```
package WebGUI::Asset::Wobject::HelloWorld;

$VERSION = "1.0.0";

use strict;
use Tie::IxHash;
use base 'WebGUI::Asset::Wobject';

#-------------------------------------------------------------------
sub definition {
   my $class = shift;
   my $session = shift;
   my $definition = shift;

```

```
    tie my %properties, 'Tie::IxHash';
    %properties = ();

    push(@{$definition}, {
        assetName=>"Hello World",
        autoGenerateForms=>1,
          tableName=>'HelloWorld',
          className=>'WebGUI::Asset::Wobject::HelloWorld',
          properties=>\%properties
    });
    return $class->SUPER::definition($session, $definition);
}

#-------------------------------------------------------------------
sub view {
    my $self = shift;
    my $session = $self->session;

    return "Hello World!";
}

#-------------------------------------------------------------------
#                    INSTALL / UNINSTALL
#-------------------------------------------------------------------

use base 'Exporter';
our @EXPORT = qw(install uninstall);
use WebGUI::Session;

#-------------------------------------------------------------------
sub install {
    my $config = $ARGV[0];
    my $home = $ARGV[1] || "/data/WebGUI";
    unless ($home && $config) {
        die "usage: perl -MWebGUI::Asset::Wobject::HelloWorld -e install
www.example.com.conf\n";
    }
    print "Installing asset.\n";
    my $session = WebGUI::Session->open($home, $config);
    $session->config->addToArray("assets","WebGUI::Asset::Wobject::HelloWorld");
    $session->db->write("create table HelloWorld (
        assetId varchar(22) binary not null,
          revisionDate bigint not null,
        primary key (assetId, revisionDate)
    )");
    $session->var->end;
    $session->close;
    print "Done. Please restart Apache.\n";
}

#-------------------------------------------------------------------
sub uninstall {
    my $config = $ARGV[0];
    my $home = $ARGV[1] || "/data/WebGUI";
    unless ($home && $config) {
        die "usage: perl -MWebGUI::Asset::Wobject::HelloWorld -e uninstall
www.example.com.conf\n";
    }
    print "Uninstalling asset.\n";
    my $session = WebGUI::Session->open($home, $config);
    $session->config->deleteFromArray("assets","WebGUI::Asset::Wobject::HelloWorld");
    my $rs = $session->db->read("select assetId from asset where
className='WebGUI::Asset::Wobject::HelloWorld'");
    while (my ($id) = $rs->array) {
        my $asset = WebGUI::Asset->new($session, $id,
"WebGUI::Asset::Wobject::HelloWorld");
        $asset->purge if defined $asset;
    }
    $session->db->write("drop table HelloWorld");
    $session->var->end;
    $session->close;
    print "Done. Please restart Apache.\n";
}



1;
```

Example: Recipe Catalog

Similar to creating a new asset, when creating a new wobject you should begin with the wobject skeleton available at /data/WebGUI/lib/WebGUI/Asset/Wobject/_NewWobject.skeleton. The wobject skeleton contains all of the methods you'll need to get your base wobject up and running.

Copy the wobject skeleton into a new Perl module that will be your new wobject. For the purposes of this chapter, create your recipe application allowing users to search for, add, edit, and delete recipes.

```
cd /data/WebGUI/lib/WebGUI/Asset/Wobject
cp _NewWobject.skeleton RecipeCatalog.pm
```

Update the first line of code to reflect the name of your wobject package WebGUI::Asset::Wobject::RecipeCatalog.

As with assets, you now need to build your definition based on the table structure for your RecipeCatalog asset. This is the exactly same process as creating the definition for assets as described in the chapter on creating assets. The database table looks like:

```
CREATE TABLE RecipeCatalog (
                assetId VARCHAR(22) BINARY NOT NULL,
                revisionDate BIGINT NOT NULL,
                viewTemplateId VARCHAR(22) BINARY NOT NULL,
                editRecipeTemplateId VARCHAR(22) BINARY NOT NULL,
                viewRecipeTemplateId VARCHAR(22) BINARY NOT NULL,
                groupToPost VARCHAR(22) BINARY NOT NULL,
                paginateAfter INTEGER DEFAULT '25',
        PRIMARY KEY (assetId,revisionDate)
);
```

After adding the assetId / revisionDate composite key, create three new templates. The first template (viewTemplateId) will be used to display a paginated list of all of the recipes in your database. The second (editRecipeTemplateId) will be used to create the form for entering new recipes. Finally, the third (viewRecipeTemplateId) will be used to view individual recipes.

Additionally, you will notice a groupToPost field. With this field you will let your administrators choose to differentiate between those who can view the recipes and those who can post them. You will also notice a paginateAfter field in which you will let your administrator designate how many recipes will be displayed before they are paginated.

From this table you can create your definition which looks like this (assume for this example that no internationalization is necessary):

```
sub definition {
        my $class = shift;
        my $session = shift;
        my $definition = shift;
        tie my %properties, 'Tie::IxHash';
                %properties = (
                paginateAfter  => {
                        fieldType      =>"integer",
                        defaultValue =>25,
                        tab               =>"properties",
                        hoverHelp    =>"Enter the number of recipes per page",
                        label            =>"Paginate Recipes After",
                },
                viewTemplateId  => {
                        fieldType      =>"template",
                        defaultValue =>'RecipeCatalog000000001',
                        tab               =>"display",
                        namespace    =>"RecipeCatalog/view",
                        hoverHelp    =>"View template for the recipe catalog",
                        label            =>"View Recipe Catalog Template",
                },
                editRecipeTemplateId => {
                        fieldType      =>"template",
                        defaultValue =>'RecipeCatalog000000002',
                        tab               =>"display",
                        namespace    =>"RecipeCatalog/edit",
                        hoverHelp    =>"View template for the recipe catalog",
                        label            =>"Edit Recipe Template",
                },
                viewRecipeTemplateId => {
                        fieldType      =>"template",
                        defaultValue =>'RecipeCatalog000000003',
                        tab               =>"display",
                        namespace   =>"RecipeCatalog/viewRecipe",
                        hoverHelp    =>"Template for viewing recipes",

                        label          =>"View Recipe Template",
                },
                groupToPost => {
                        fieldType      =>"group",
                        defaultValue =>3,
                        tab               =>"security",
                        hoverHelp    =>"Choose the group to post recipes",
                        label       =>"Group To Post",
                }
        );

        push(@{$definition}, {
                assetName            =>"Recipe Catalog",
                autoGenerateForms =>1,
                tableName            =>'RecipeCatalog',
                className            =>'WebGUI::Asset::Wobject::RecipeCatalog',
                properties           =>\%properties
         });
         return $class->SUPER::definition($session, $definition);
}
```

Notice a few things. All of the default template variables are hardcoded. This is because the default templates will be included with the installation method of the asset. Each template also has a separate namespace. This ensures that the right templates will always be chosen and you won't wind up with a page that has the wrong template for the variables returned.

Remember, asset ID's must be exactly 22 characters long.

Installing Your Wobject

Once the definition is created, it is useful to get your wobject on your WebGUI instance so you can see progress as you develop. It is unrealistic to think you can build an entire application without ever looking at it, so getting it on the site is a crucial early step.

As with developing assets, _NewWobject.skeleton comes with an install method which performs all of the installation necessary for your asset, such as adding the wobject to the config file and creating all of the wobject database tables.

There is one additional step you should add to your install method that is not included in the _NewWobject.skeleton class, which is importing all of the default templates for your asset. This not only saves the user the headache of having to manually update your templates, but it also ensures that you are able to hard code the default template values into your

See Appendix 1 for more information about Advanced Install Scripts.

definition and database tables as you saw earlier in the chapter.

To get a visual working environment for building your application, update the install method of your asset and run the script. Once this is done, you should be able to turn admin on in WebGUI and see your asset in the New Content menu of the Admin Bar.

This is a good time to note a few things. First, the installation method is error prone. It is wise to also update the uninstall method in case you make mistakes. That way you can uninstall and start over. Second, installing this early in the application process typically means that you are going to have to make changes. Even though you did some preliminary design, there is usually something that comes up during the development phase that you didn't think of and requires changes to templates and database tables.

There are two ways to approach this problem. The first is simply to embed everything in your install scripts and any time you have to make a change to a database table or template, uninstall the application and reinstall it. This will ensure that your scripts continue to work and that you haven't made any syntax errors that might cause it to fail. The drawback is that every time you make a mistake that affects a template or the database you lose any test data you had. Given that during the development process you will be changing templates on a regular basis, this solution can be tedious and time consuming.

A better way to manage the installation process is to move all of the templates outside of the script and create an additional executable method which simply modifies all of the templates. This way, after the initial install you can simply run your new method if you make a change to the template. Additionally, your templates (which are mostly HTML) now exist in separate files making them easier to edit and able to be part of any versioning system you use. A cookbook chapter at the end of this book further describes how to do this. For the rest of this chapter, simply assume that your install script was generated through the development process and is now complete, thus it will not be demonstrated here.

```
sub install {
        my $config    = $ARGV[0];
        my $home      = $ARGV[1] || "/data/WebGUI";
        my $className = "WebGUI::Asset::Wobject::RecipeCatalog";
        unless ($home && $config) {
                die "usage: Perl -M$className -e install yoursite.conf\n";
        }
```

```
print "Installing asset.\n";
my $session = WebGUI::Session->open($home, $config);
#Add wobject to config file
$session->config->addToArray("assets",$className);
#Create database tables
$session->db->write("CREATE TABLE RecipeCatalog (
        assetId VARCHAR(22) BINARY NOT NULL,
        revisionDate BIGINT NOT NULL,
        viewTemplateId VARCHAR(22) BINARY NOT NULL,
        editRecipeTemplateId VARCHAR(22) BINARY NOT NULL,
        viewRecipeTemplateId VARCHAR(22) BINARY NOT NULL,
        groupToPost VARCHAR(22) BINARY NOT NULL,
        paginateAfter INTEGER DEFAULT '25',
        PRIMARY KEY (assetId,revisionDate)
)");
$session->db->write("CREATE TABLE RecipeCatalog_recipe (
        recipeId VARCHAR(22) BINARY NOT NULL,
        assetId VARCHAR(22) BINARY NOT NULL,
        title VARCHAR(100) NOT NULL,
        description TEXT NOT NULL,
        servings VARCHAR(30) NOT NULL,
        ingredients TEXT NOT NULL,
        directions TEXT NOT NULL,
        PRIMARY KEY (recipeId)
)");
### Create a folder asset to store the default templates
my $importNode = WebGUI::Asset->getImportNode($session);
my $newFolder = $importNode->addChild({
        className   => "WebGUI::Asset::Wobject::Folder",
        title       => "Recipe Catalog",
        menuTitle   => "Recipe Catalog",
        url         => "recipe_catalog_folder",
        groupIdView => "3"
},"RecipeCatalogFolder001");
#Create the templates
#Recipe Catalog View Template
my $recipeCatalogTmpl = q|
<tmpl_if session.var.adminOn>
        <p><tmpl_var controls></p>
</tmpl_if>
<tmpl_if displayTitle>
        <h2><tmpl_var title></h2>
</tmpl_if>
```

```
<tmpl_if error_msg>
     <div class="error"><tmpl_var error_msg></div>
</tmpl_if>
<tmpl_if canPost>
    <a href="<tmpl_var add_recipe>">add a recipe</a>
</tmpl_if>
<tmpl_if has_recipes>
    <table border="1" cellpadding="3" cellspacing="0">
        <tbody>
            <tr>
                <th>Recipe Name</th>
                <tmpl_if canPost>
                <th>&nbsp;</th>
                <th>&nbsp;</th>
                </tmpl_if>
            </tr>
            <tmpl_loop recipe_loop>
            <tr>
                <td>
                    <a href="<tmpl_var view_recipe>">
                        <tmpl_var recipeName>
                    </a>
                </td>
                <tmpl_if canPost>
                <td>
                    <a href="<tmpl_var edit_recipe>">edit</a>
                </td>
                <td>
                    <a href="<tmpl_var delete_recipe>">delete</a>
                </td>
                </tmpl_if>
            </tr>
            </tmpl_loop>
        </tbody>
    </table>
    <tmpl_if pagination.pageCount.isMultiple>
        <tmpl_unless pagination.isFirstPage>
            <tmpl_var pagination.firstPage>
        </tmpl_unless>
        <tmpl_var pagination.previousPage>
        <tmpl_var pagination.nextPage>
        <tmpl_unless pagination.isLastPage>
            <tmpl_var pagination.lastPage>
```

```
                </tmpl_unless>
        </tmpl_if>
<tmpl_else>
        <table border="1" cellpadding="3" cellspacing="0">
                <tbody>
                        <tr>
                                <td>No Recipes Have Been Entered.</td>
                        </tr>
                </tbody>
        </table>
</tmpl_if>
|;
#Recipe Edit Template Code
my $recipeEditTmpl = q|
<tmpl_if error_msg>
        <div class="error"><tmpl_var error_msg></div>
</tmpl_if>
        <h3>Edit Recipe</h3>
<tmpl_var form_header>
<tmpl_var form_hidden>
<table border="0" cellpadding="3" cellspacing="0">
        <tbody>
                <tr>
                        <td>Recipe Name</td>
                        <td><tmpl_var form_title></td>
                </tr>
                <tr>
                        <td>Description</td>
                        <td><tmpl_var form_description></td>
                </tr>
                <tr>
                        <td>Servings</td>
                        <td><tmpl_var form_servings></td>
                </tr>
                <tr>
                        <td>Ingredients</td>
                        <td><tmpl_var form_ingredients></td>
                </tr>
                <tr>
                        <td>Directions</td>
                        <td><tmpl_var form_directions></td>
                </tr>
                <tr>
```

```
                        <td colspan="2" align="right"><tmpl_var form_submit></td>
                </tr>
        </tbody>
</table>
<tmpl_var form_footer>
|;
#Recipe View Template Code
my $recipeViewTmpl = q|
<h3><tmpl_var recipe_name></h3>
<br /><br />
<table border="0" cellpadding="3" cellspacing="0">
        <tbody>
                <tr>
                        <td>Description</td>
                        <td><tmpl_var recipe_description></td>
                </tr>
                <tr>
                        <td>Servings</td>
                        <td><tmpl_var recipe_servings></td>
                </tr>
                <tr>
                        <td>Ingredients</td>
                        <td><tmpl_var recipe_ingredients></td>
                </tr>
                <tr>
                        <td>Directions</td>
                        <td><tmpl_var recipe_directions></td>
                </tr>
        </tbody>
</table>
<br /> <br />
<a href="<tmpl_var home_url>">back to recipe catalog</a>
|;
#Add the templates to the folder
$newFolder->addChild({
        className   =>"WebGUI::Asset::Template",
        ownerUserId =>'3',
        groupIdView =>'7',
        groupIdEdit =>'12',
        title       =>"Default Recipe Catalog Template",
        menuTitle   =>"Default Recipe Catalog Template",
        url         =>"default_recipe_catalog_template",
        namespace   =>"RecipeCatalog/view",
		template	=>$recipeCatalogTmpl,
		},'RecipeCatalog000000001'
	);
	$newFolder->addChild({
		className=>"WebGUI::Asset::Template",
		ownerUserId=>'3',
		groupIdView=>'7',
		groupIdEdit=>'12',
		title=>"Default Edit Recipe Template",
		menuTitle=>"Default Edit Recipe Template",
		url=>"default_edit_recipe_template",
		namespace=>"RecipeCatalog/edit",
		template=>$recipeEditTmpl,
		},'RecipeCatalog000000002'
	);
	$newFolder->addChild({
		className=>"WebGUI::Asset::Template",
		ownerUserId=>'3',
		groupIdView=>'7',
		groupIdEdit=>'12',
		title=>"Default View Recipe Template",
		menuTitle=>"Default View Recipe Template",
		url=>"default_view_recipe_template",
		namespace=>"RecipeCatalog/viewRecipe",
		template=>$recipeViewTmpl,
		},'RecipeCatalog000000003'
		);
	#Commit the working version tag
	my $workingTag	= WebGUI::VersionTag->getWorking($session);
	my $workingTagId = $workingTag->getId;
	my $tag = WebGUI::VersionTag->new($session,$workingTagId);
	if (defined $tag) {
		print "Committing tag\n";
		$tag->set({comments=>"Folder created by Asset Install Process"});
		$tag->requestCommit;
	}
	$session->var->end;
	$session->close;
	print "Done. Please restart Apache.\n";
}
```

The following collateral table has been added for storing recipes (will discuss this more a bit later in the chapter).

```
CREATE TABLE RecipeCatalog_recipe (
        recipeId VARCHAR(22) BINARY NOT NULL,
                assetId VARCHAR(22) BINARY NOT NULL,
                title VARCHAR(100) NOT NULL,
                description TEXT NOT NULL,
                servings VARCHAR(30) NOT NULL,
                ingredients TEXT NOT NULL,
                directions TEXT NOT NULL,
          PRIMARY KEY (recipeId)
);
```

Additionally, a folder has been created and added to the import node. WebGUI's import node can easily become cluttered with templates if you are not responsible about where you put things. For that reason, it's important to create a folder to store the content related to your asset. Not only does it reduce clutter, but it makes it easier to find asset related material.

A few changes have also been made to the uninstall script:

```
        sub uninstall {
                my $config    = $ARGV[0];
                my $home      = $ARGV[1] || "/data/WebGUI";
                my $className = "WebGUI::Asset::Wobject::RecipeCatalog";
                unless ($home && $config) {
                        die "usage: Perl -M$className -e uninstall yoursite.conf\n";
                }
                print "Uninstalling asset.\n";
                my $session = WebGUI::Session->open($home, $config);
                #Delete wobject from config file
                $session->config->deleteFromArray("assets",$className);
                #Delete all assets and default templates
                my $rs = $session->db->read(qq{
                        select
                                assetId
                        from
                                asset
                        where
                                className='$className'
                                or assetId like 'RecipeCatalog%'
                });
                while (my ($id) = $rs->array) {
                        my $asset = WebGUI::Asset->new($session, $id, $className);
                        $asset->purge if defined $asset;
                }
                #Drop asset related tables
                $session->db->write("drop table if exists RecipeCatalog");
                $session->db->write("drop table if exists RecipeCatalog_recipe");
                $session->var->end;
                $session->close;
                print "Done. Please restart Apache.\n";
        }
```

Notice that you are additionally getting rid of your default templates by adding the “or assetId like 'RecipeCatalog%'” clause to the query retrieving all of the assets to purge. Additionally, you are dropping the collateral table as well as the main wobject table.

One additional note. In order to display the wobject properly, you need to update the prepareView method to prepare the correct default template. In the wobject skeleton, it assumes a template variable of tempalteId.

```
sub prepareView {
        my $self = shift;
        $self->SUPER::prepareView();
        my $template = WebGUI::Asset::Template->new(
                $self->session,
                $self->get("viewTemplateId")
        );
        $template->prepare;
        $self->{_viewTemplate} = $template;
}
```

If you fail to do this, WebGUI will complain that it can't instantiate your template.

Building Your Wobject

Once the base wobject is installed you should be able to see it in the New Content menu of the Admin Bar and add it to the site. The edit screen should work correctly, and you should see whatever default template you have for the view when you save. Since this is exactly the same process that is described in the chapter on building assets, you are simply going to point out a few additional things in the view method.

```
sub view {
    my $self    = shift;
    my $session = $self->session;
    my $db      = $session->db;
    my $user    = $session->user;
    my $var     = $self->get;
    my $canPost = $user->isInGroup($self->get("groupToPost"));
    #Create a new instance of a paginator
    my $p = WebGUI::Paginator->new(
        $session,
        $self->getUrl,
        $self->get("paginateAfter"),
    );
    #Create a query to retrieve all of the recipes
    my $sql = q|
        SELECT
            *
        FROM
            RecipeCatalog_recipe
        WHERE
            assetId=?
        ORDER BY
            title ASC
    |;
    #Pass the query and arguments to the paginator
    $p->setDataByQuery($sql,undef,undef,[$self->getId]);
    #Use the Paginator to get back an array ref of the current page of data
    my $pageData = $p->getPageData;
    #Create an array to store the recipes
    my @recipeLoop = ();
    foreach my $row (@{$pageData}) {
        #Create a hashref to store individual loop data
        my $recipeHash = {};
        #Store all relevant recipe data
        $recipeHash->{'recipeName' } = $row->{title};
        $recipeHash->{'recipeDesc' } = $row->{description};
        $recipeHash->{'servings'   } = $row->{servings};
         $recipeHash->{'ingredients'} = $row->{ingredients};
        $recipeHash->{'direction'  } = $row->{directions};
```

```
            #Create a link to view the recipe
             $recipeHash->{'view_recipe'} = $self->getUrl(
                    "func=viewRecipe;recipeId=".$row->{recipeId}
            );
            #Create edit and delete links
             if($canPost) {
                    $recipeHash->{'edit_recipe'  } = $self->getUrl(
                            "func=editRecipe;recipeId=".$row->{recipeId}
                    );
                    $recipeHash->{'delete_recipe'} = $self->getUrl(
                            "func=deleteRecipe;recipeId=".$row->{recipeId}
                    );
            }
            #Push the hashref onto the recipe loop
            push(@recipeLoop,$recipeHash);
    }
    #Create a template variable for the recipe loop
    $var->{'recipe_loop'} = \@recipeLoop;
    #Append pagination template variables
    $p->appendTemplateVars($var);
    #Create a template variable in case there is no data
    $var->{'has_recipes'} = (scalar(@recipeLoop) > 0);
    #Show a link to add a recipe if the user can post new recipes
    if($canPost) {
            $var->{'canPost'   } = "true";
            $var->{'add_recipe'} = $self->getUrl("func=editRecipe");
    }
     #Return the template to the www_view method
    return $self->processTemplate($var, undef, $self->{_viewTemplate});
}
```

In the first few lines set up some helper variables for things contained in session like the database and user object. Then, create a hash reference, $var, in which to put all of your template variables. Calling $self->get initially populates the hash reference with all of the superclass asset data.

Then, instantiate a Paginator object to automate pagination of your asset. The Paginator API is extremely useful when building assets that list data. Not only does it provide you a simple means to allow users to page through large data sets, but it does so efficiently.

Create a new Paginator instance by calling the constructor and passing it the current session object, the URL that you wish to have paginated (the link to your data set), and the number of data items you wish to have per page.

Once you've created a Paginator instance, you can then pass either a query (most efficient) or an array of data (least efficient) to paginate. The Paginator will calculate the number of items to return, as well as the current page, and return you only the relevant data.

Additionally, the Paginator has methods for easily appending template variables to your template for displaying pagination. For more information about the Paginator see the WebGUI::Paginator API.

Calling $p->getPageData returns a full page of data which you then need to export to your HTML Template. Because you don't know exactly how many rows of data will be returned at any given time, you need to return it as a loop which can be iterated over inside the template. Do this by creating an array reference of hash references. Each hash reference represents a single iteration in the loop and contains the same keys as the previous iteration.

When you assign the array reference to a template variable you create a loop which can be referenced from within your template using the <tmpl_loop> structure.

Finally, return the template to the www_view method for the wobject. The www_view method contains the logic to wrap your template in the page style and does this for you automatically.

www_ methods

By prefacing a method with www_, you tell WebGUI that this method should be able to be called from the website. You've seen two such methods so far, www_view and www_edit, in the chapter on writing assets. WebGUI calls these methods automatically, so it might not be clear how this functionality works. To call any www_ method in WebGUI, you simply add func=methodName to the end of a URL where methodName is whatever follows www_. For instance, if a method called www_showDetails was created in the wobject, it would be called by simply appending func=showDetails to the end of the asset's URL: http://www.mysite.com/myasset?func=showDetails. This tells WebGUI that you don't wish to see the default view of the wobject, but rather the showDetails page.

Whenever you create a wobject, you will likely need to create your own www_ methods for various tasks you would like your users to do. For instance, in the recipe application, in order to create a new recipe, you are going to need to provide your users a form for adding new recipes. You can accomplish this task by creating a www_editRecipe method.

```
sub www_editRecipe {
        my $self      = shift;
        my $session   = $self->session;
        my $db        = $session->db;
        my $user      = $session->user;
        my $form      = $session->form;
        my $privilege = $session->privilege;
        my $var       = $self->get;
        my $canPost   = $user->isInGroup($self->get("groupToPost"));
        #Create an error template variable if an error happened
        $var->{'error_msg'} = shift;
        #Only allow those who can post recipes to post.
        return $privilege->insufficient() unless ($canPost);
        #Get the collateral row if one exists
        my $recipe    = $self->getCollateral(
                "RecipeCatalog_recipe",
                "recipeId",
                $form->get("recipeId")
        );
        #Create the form elements
        $var->{'form_header'} = WebGUI::Form::formHeader($session,{
                action=>$self->getUrl()
        });
        $var->{'form_hidden'} = WebGUI::Form::Hidden($session,{
                name  =>"func",
                value =>"editRecipeSave"
        });
        $var->{'form_hidden'} .= WebGUI::Form::Hidden($session,{
                name  =>"recipeId",
                value =>$recipe->{recipeId}
        });
        $var->{'form_title'} = WebGUI::Form::Text($session,{
                name      =>"title",
                value     =>$form->get("title") || $recipe->{title},
                maxLength => 100
        });
        my $description
```

```
                = $form->process("description","HTMLArea") || $recipe->{description};
        $var->{'form_description'} = WebGUI::Form::HTMLArea($session,{
                name  =>"description",
                value =>$description,
        });
        $var->{'form_servings'} = WebGUI::Form::Text($session,{
                name  =>"servings",
                value =>$form->get("servings") || $recipe->{servings}
        });
        my $ingredients
                = $form->process("ingredients","HTMLArea") || $recipe->{ingredients};
        $var->{'form_ingredients'} = WebGUI::Form::HTMLArea($session,{
                name  =>"ingredients",
                value =>$ingredients,
        });
        my $directions
                = $form->process("directions","HTMLArea") || $recipe->{directions};
        $var->{'form_directions'} = WebGUI::Form::HTMLArea($session,{
                name  =>"directions",
                value =>$directions,
        });
        $var->{'form_submit'} = WebGUI::Form::submit($session);
        $var->{'form_footer'} = WebGUI::Form::formFooter($session);
        #return the template wrapped in the page style.
        my $templateId = $self->getValue("editRecipeTemplateId");
        my $template   = $self->processTemplate($var,$templateId);
        return $self->processStyle($template);
}
```

Notice that this looks very similar to your view method. There are a few differences that are important to note. First of all, it is important to understand that you are fully responsible for security on your own pages. WebGUI takes care of security when it comes to viewing the wobject, but if you are going to create your own web accessible methods, you need to make sure you set security accordingly. You'll notice that the example uses WebGUI's privilege API to block anyone that isn't in the groupToPost group set in the wobject properties from seeing the page.

The other major difference is that you cannot simply return $self->processTemplate() if you want the contents of your template to be wrapped in the site's style. Remember earlier in the chapter it was said that one of the drawbacks to legacy widgets and wobjects was that they didn't enable you to send raw content back to the browser. This is where you have that power. WebGUI's wobject API has a method which wraps the style around the template for you called processStyle():

```
$wobject->processStyle($template)
```

Keep in mind that processStyle will not process the template for you. You must do this prior to calling this method. If you do want to return raw content (such as an RSS feed), you simply omit the style wrapper and return the results of $self->processTemplate (which comes from the Asset API).

Another wobject API method used in the www_editRecipe method is the getCollateral method. getCollateral is the first of several methods which make it easy to deal with collateral data in your wobjects:

```
my $row = $wobject->getCollateral(collateralTable,keyName,keyValue)
```

This method returns one row of data from the collateral table you specify by looking for the key which has the value passed in. Some of the other collateral methods will be covered later in this chapter. Here are a few that will not be discussed in detail.

- $wobject->moveCollateralDown(collateralTable, keyName, keyValue); This method reorders content that is sequenced in the database switching its position with the next row.
- $wobject->moveCollateralUp(collateralTable,keyName,keyValue) This method reorders content that is sequenced in the database switching it's position with the previous row.
- $wobject->reorderCollateral(collateralTable,keyName) When working with sequential data, it is sometimes necessary to fill gaps in the sequencing that might occur from removing data. This method does this for you automatically.

Finally, notice that in the form you have hidden a field called func which has a value of editRecipeSave. Since this is a form post, you need another method which processes your data and puts it into the database. Do this by creating another www_ method and including a hidden func variable which tells WebGUI where to send the post content.

```
sub www_editRecipeSave {
        my $self      = shift;
        my $session   = $self->session;
        my $db        = $session->db;
        my $user      = $session->user;
         my $form      = $session->form;
        my $privilege = $session->privilege;
        my $var       = $self->get;
        my $canPost   = $user->isInGroup($self->get("groupToPost"));
        #Only allow those who can post recipes to post.
        return $privilege->insufficient() unless ($canPost);
        #Build a list of properties based on form elements passed in
        my $props = {};
        $props->{'recipeId'   } = $form->process("recipeId","hidden");
        $props->{'title'      } = $form->process("title","text");
        $props->{'description'} = $form->process("description","HTMLArea");
        $props->{'servings'   } = $form->process("servings","text");
        $props->{'ingredients'} = $form->process("ingredients","HTMLArea");
        $props->{'directions' } = $form->process("directions","HTMLArea");
        #Do some error checking.  Return the user to
        if($props->{'title'} eq "") {
                return $self->www_editRecipe(
                         "All recipes must have a title"
                );
        }
        if($props->{'description'} eq "") {
                return $self->www_editRecipe(
                        "All recipes must have a description"
                );
        }
        if($props->{'servings'} eq "") {
                return $self->www_editRecipe(
                        "All recipes must have the number of servings"
                );
        }
        if($props->{'ingredients'} eq "") {
                return $self->www_editRecipe(
                        "All recipes must list ingredients"
                );
        }
        if($props->{'directions'} eq "") {
                return $self->www_editRecipe(
                        "All recipes must have cooking directions"
                );
        }

        #Add or update the database with the recipe data
        $self->setCollateral(
                "RecipeCatalog_recipe",
                "recipeId",
                $props,
                0,
                1
        );
        #Return the default view of the wobject.
        return "";
}
```

Like before, this method begins by checking privileges. A common mistake for new developers is to add privilege checks to the view pages but not to the submit pages, making it possible for users to post content to pages they don't have access to.

Also note that another collateral method, setCollateral, is used. This method is invaluable when it comes to writing wobjects because it saves you a lot of steps. Normally, when you write software that has to insert and update to a database table, you wind up with code that first has to check to make sure that the content doesn't already exist in the database and then decide whether to perform an insert or an update. The setCollateral method of WebGUI's wobject API does this for you. Additionally, it automates adding assetIds and sequence numbers to your collateral tables:

```
$wobject->setCollateral( collateralTable, keyName, properties, sequenceNumber, assetId)
```

This method accepts a hash reference of properties to be inserted or updated in a database table. If a value is found in the properties hash reference for keyName, it updates all columns in the database matching the key of the properties hash reference with the associated value. If no keyValue is found, or the keyValue is the string “new,” it will insert all of the data in the properties hash reference. This is a huge benefit as any developer who has dealt with this before can attest.

Additionally, setCollateral will automatically sequence tables with the column name sequenceNumber. You must be careful to exactly name your column this to take advantage of auto sequencing. Finally, setCollateral will automatically update an assetId column if you specify. The column name must be “assetId” for this to work properly.

One final thing to note about this method is that it returns an empty string. This tells WebGUI to return the default view for the asset.

Container Wobjects

This is a good opportunity to digress and talk briefly again about Wobject design. As you've seen, WebGUI's wobject API provides you with methods for dealing with collateral data. However, there are many times when your collateral data needs to be indexed for searching, versioned, or maintained in a tree like structure. When you run into these roadblocks, it is a good idea to think about creating a container wobject.

A container wobject does not contain any collateral data. Instead, it is a parent for other assets for which it is responsible for displaying. WebGUI is full of container assets. The most notable ones being the Collaboration System, which is responsible for displaying Thread and Post assets; the Calendar asset, which is responsible for displaying Event assets; and the Page Layout asset, which is responsible for displaying all of its children.

This works well in situations described above because WebGUI automatically indexes every asset. What this means is that if you are writing a wobject in which you need to search for collateral data, by creating a new asset type for that asset and a container wobject, you won't have to create a search. You will simply use WebGUI's default search to get the job done.

Also, you may need to version your data. This can be tedious and time consuming if you attempt to do this manually within the wobject. Instead, if you create a new asset type and container wobject, any changes made to your sub-assets will automatically be versioned by WebGUI.

This is a very powerful method employed for developing wobjects of this type. The recipe asset could very well have taken a different route in the design phase and become a container wobject which displays Recipe Assets.

Finishing Up

Now that your users can view, create, and edit recipes, you need to polish off the wobject by adding methods to let the users view and delete recipes. Finally, you need to update the purge and duplicate methods to account for your collateral data.

Start by creating a www_veiwRecipes method which allows users to view the recipes that have been entered:

```
sub www_viewRecipe {
        my $self      = shift;
        my $session   = $self->session;
        my $db        = $session->db;
        my $user      = $session->user;
        my $form      = $session->form;
        my $privilege = $session->privilege;
        my $var       = $self->get;
        #Only allow those who can view the wobject to view the recipes.
        return $privilege->insufficient() unless ($self->canView);
        my $recipeId = $form->get("recipeId");
        #Get the collateral row if one exists.
        my $recipe    = $self->getCollateral(
                "RecipeCatalog_recipe",
                "recipeId",
                $form->get("recipeId")
        );
        #Don't let users try to view recipes that don't exist
        return "" if($recipe->{recipeId} eq "new");
        #Create template variables for the recipe data
        $var->{'recipe_name'       } = $recipe->{title};
        $var->{'recipe_description'} = $recipe->{description};
        $var->{'recipe_servings'   } = $recipe->{servings};
        $var->{'recipe_ingredients'} = $recipe->{ingredients};
        $var->{'recipe_directions' } = $recipe->{directions};
        #Create a url for a "back" button
        $var->{'home_url'} = $self->getUrl();
        #return the template wrapped in the page style.
        my $templateId = $self->getValue("viewRecipeTemplateId");
        my $template   = $self->processTemplate($var,$templateId);
        return $self->processStyle($template);
}
```

This method looks very similar to the others. You'll note that one slight change has been made in that you are checking the view privileges of the wobject to determine whether or not a user can view the recipe. This differs from the the previous methods which were looking to see if a user could post. Also note that you use the getCollateral and processStyle methods from the wobject superclass again.

Next, add a way for users who can post to delete recipes. You could certainly extend this to allow only admins or users who own the posts to delete the records, but for the purposes of this example simply allow anyone who can post to delete any recipe.

```
sub www_deleteRecipe {
	my $self      = shift;
	my $session   = $self->session;
	my $user      = $session->user;
	my $form      = $session->form;
	my $privilege = $session->privilege;
	my $var       = $self->get;
	my $canPost   = $user->isInGroup($self->get("groupToPost"));
	#Only allow those who can post recipes to post.
	return $privilege->insufficient() unless ($canPost);
	my $recipeId  = $form->get("recipeId");
	#Don't let users try to delete recipes that don't exist
	return "" unless ($recipeId);

	my $message = "Are you sure you want to delete this recipe?";
	my $yesUrl  = $self->getUrl(
		"func=deleteRecipeConfirm;recipeId=$recipeId"
	);
	return $self->processStyle($self->confirm($message,$yesUrl));
}
```

This method uses another wobject API method used for displaying a confirmation page to the user.

```
$html = $wobject->confirm($message,$yesUrl,$noUrl)
```

This method returns a standard, pre-configured confirmation screen which allows users to change their minds if they accidentally delete the wrong item. Passing in a message, along with a URL to go to if the user would like to continue deleting, and a URL to go to if a mistake was made returns standard HTML used throughout WebGUI for confirming user intentions. If you do not supply a noUrl, the users will be taken back to the main view of the application if they make a mistake.

The example above indicates that if the user is purposely deleting the content, he/she should be directed to the www_deleteRecipeConfirm page to which you pass the unique key of the table in which recipes are stored.

```
sub www_deleteRecipeConfirm {
	my $self      = shift;
	my $session   = $self->session;
	my $user      = $session->user;
	my $form      = $session->form;
	my $privilege = $session->privilege;
	my $var       = $self->get;
	my $canPost   = $user->isInGroup($self->get("groupToPost"));
	 #Only allow those who can post recipes to post.
	return $privilege->insufficient() unless ($canPost);
	my $recipeId  = $form->get("recipeId");
	#Don't let users try to delete recipes that don't exist
	return "" unless ($recipeId);
	$self->deleteCollateral(
		"RecipeCatalog_recipe",
		"recipeId",
		$recipeId
	);
	return "";
}
```

This method uses the final wobject API method discussed in this chapter.

```
$wobject->deleteCollateral(collateralTable,keyName,keyValue);
```

This method deletes a single row from the collateral table where keyName matches keyValue.

The last thing you need to do for this wobject to be complete is to update the purge and duplicate methods that were discussed in the chapter on writing assets. These methods are slightly different for wobjects as you need to account for collateral data.

```
sub purge {
	my $self = shift;
	my $db   = $self->session->db;
	my $sql = q{
		DELETE
		FROM
			RecipeCatalog_recipe
		WHERE
			assetId=?
	};]
	$db->write($sql,[$self->getId]);
	return $self->SUPER::purge;
}
```

When purge is called on an instance of a wobject, it needs to remove all collateral data associated with the wobject. This goes back to your design. A common mistake is to forget that multiple instances of wobjects can exist across the site. It is important to make sure that all collateral is related to its asset index so you can properly purge as well as duplicate.

```
	sub duplicate {
	my $self = shift;
	my $db   = $self->session->db;
	my $newAsset = $self->SUPER::duplicate(@_);
	my $sql      = q{
		SELECT
			*
		FROM
			RecipeCatalog_recipe
		WHERE
			assetId=?
	};
	my $sth = $db->read($sql,[$self->getId]);
	while (my $row = $sth->hashRef) {
		$row->{recipeId} = "new";
		$row->{assetId } = $newAsset->getId;
		$newAsset->setCollateral(
			"RecipeCatalog_recipe",
			"recipeId",
			$row,
			0,
			1
```

```
                );
        }
        return $newAsset;
}
```

The duplicate method selects all of the data related to the original asset, and copies it using the setCollateral method of the new asset. This ensures that all the recipes, as well as the asset itself, are copied.

Wobject Inheritance

Finally, it's worth some time to talk about wobject inheritance. As said at the beginning of the chapter, there are times where you want an existing asset to work just a bit differently. In cases like this, you can inherit all of the properties of the existing wobject and simply override the parts you want to change.

Say, for example, that you would like to export a few more template variables in the view of your recipe wobject. Rather than starting from scratch or copying the recipe wobject exactly, you can easily extend the wobject by using it as your base instead of the wobject class.

```
use base 'WebGUI::Asset::Wobject::RecipeCatalog';
```

Since the RecipeCatalog wobject is a wobject, your new wobject is one also. Like every wobject, you will need to create a database table that has the assetId / revisionDate composite key as well as a definition. The good news is, if you aren't changing any of the properties, your definition is going to be rather short.

Finally, to change the view, simply copy the view from the RecipeCatalog wobject into the new one and make the changes that are necessary. No collateral tables would be necessary as all of the recipes would be stored in the original RecipeCatalog tables. Instead of all the work that went into creating the new asset, overriding two methods allows you to create a brand new asset with its own functionality.

Internationalization

The examples in this chapter reference the asset written in the Assets chapter.

A major feature of WebGUI is the Internationalization (i18n) of the interface. By storing English strings in a data structure with a key, you can use the same key to store strings in Dutch, German, Spanish, and other languages. Using a combination of WebGUI::International objects in your asset, and ^International(); macros in your templates, you can make easily translatable interfaces. This example will make an i18n for the InfoCard asset created in the Assets chapter.

To use internationalization in your code, first make an i18n namespace in lib/WebGUI/i18n/English/. You can call yours Asset_InfoCard to follow with convention (assets start with Asset_, form controls start with Form_, macros with Macro_, etc...).

```
package WebGUI::i18n::English::Asset_InfoCard;
use strict;
our $I18N = {
        'assetName' => {
                message     => q{InfoCard},
                lastUpdated => 0,
                context     => q{The name of the InfoCard asset},
        },
};
1;
```

I18n modules are just Perl modules with a single “our” variable called $I18N. This variable is a hash reference of hash references, as outlined above. The key will be used to access your message in line 5, which is the internationalized text. I18n also provides for keeping track of the time lastUpdated, which is used by the i18n utility at http://i18n.webgui.org, and the context in line 7, which is how the message will be used. Sometimes the translator won't be able to figure out what he's translating from the message, which is why the context can be useful.

Once you have a shell of an i18n module, you need a way to get at it from your asset. For this, make an i18n method in your asset class.

```
	sub i18n {
		my $class	= shift;
		my $session= shift;
		return WebGUI::International->new( $session, "Asset_InfoCard" );
	}
```

Notice that this is a class method, because you'll need to be able to use it from other class methods. It takes a single argument, a WebGUI::Session object, and returns a WebGUI::International object with your i18n namespace. Now that you have a way to get at your i18n, you need to use it.

To start, revisit the definition. Remember your label and hoverHelp keys? Now you can internationalize them instead of hard-coding English, like so:

```
	sub definition {
		my $class		= shift;
		my $session= shift;
		my $definition	= shift;
		my $i18n		= __PACKAGE__->i18n( $session );

		tie my %properties, 'Tie::IxHash', (
			templateIdView => {
				tab			=> 'display',
				fieldType		=> 'template',
				label			=> $i18n->get('property templateIdView 
label'),
				hoverHelp		=> $i18n->get('property templateIdView 
description'),
				namespace		=> 'InfoCard',
			},
```

First, get your WebGUI::International object. Then, in place of literal strings, use the get method to get the appropriate string from your Asset_InfoCard i18n file (lines 11-12). Notice the key used. Property templateIdView label is descriptive and follows the general<->specific rules outlined *by Perl Best Practices.* “property” because it's an Asset property, templateIdView'is the property, and “label” is what's being displayed. Now, when these keys are alphabetized (by the http://i18n.webgui.org application) they will follow a logical pattern and be easier to translate.

Once you've changed all of your labels and hoverHelp to be i18n values, you can go back and add your English translation to your i18n module.

```
   'assetName' => {
      message     => q{InfoCard},
      lastUpdated => 0,
      context     => q{The name of the InfoCard asset},
   },
   'property templateIdView label' => {
      message     => q{View Template},
      lastUpdated => 0,
      context     => q{Label for asset property},
   },
   'property templateIdView description' => {
      message     => q{The template to view the InfoCard},
      lastUpdated => 0,
      context     => q{Description of asset property},
   },
```

Now you can do this for other English text inside of your module, and you're left with only your template having plain English labels. To make those translatable, you can use the International macro inside of your template.

The International macro allows you to perform i18n lookups anywhere a macro can be used. Like the WebGUI::International->get method, the International macro is quite easy to use.

```
<h2><tmpl_var name></h2>
<dl>
   <dt>^International('property name label','Asset_InfoCard');</dt>
   <dd><tmpl_var name></dd>

   <dt>^International('property address label','Asset_InfoCard');</dt>
   <dd><tmpl_var address> <br /> <tmpl_var city>, <tmpl_var state></dd>
</dl>
```

Notice in the sample template above, international labels are reused from your asset's edit form. This is a good idea in most cases, but sometimes you'll want to use a different label. Take your address above. In the InfoCard edit form, Address refers only to the person's street address, but in your template, you bunch the entire address together: street, city, and state. So instead, give it a different i18n label.

```
   <dt>^International('template address label', 'Asset_InfoCard');</dt>
   <dd><tmpl_var address> <br /> <tmpl_var city>, <tmpl_var state></dd>
```

And, of course, add your new label to your English i18n translation.

```
   'template address label' => {
      message     => q{Full Address},
      lastUpdated => 0,
      context     => q{Label for the address, city, and state in the template.},
   },
```

Placeholders and i18n

Most i18n values just work in all languages. You can translate words, phrases, or paragraphs and things come out just fine. Sometimes, however, the order of things must be reversed. This is most readily apparent when translating labels for numbers. How can you make i18n for, “Viewing results 1-10 of 30 total,” when those numbers are given in template variables?

Perl already has a method of dealing with such things: formats. Since formats are just strings with special parts (placeholders), you can store the format as an i18n value and use Perl's sprintf() function to replace the placeholders with the correct values.

```
my $currentPage = “1-10”;
my $total       = “50”;
sprintf “Viewing results %s of %s total”, $currentPage, $totalResults;
```

Now, the string, “Viewing results %s of %s total,” will be translated into, “Viewing results 1-10 of 50 total.” All you need to do is move the string out to one of your i18n modules.

More information on how to use formats and placeholders is available using p*erldoc -f sprintf.*

If you're in a template and you need to fill in a placeholder, all you need to do is pass it as another argument to the ^International; macro.

```
^International(“search title”, “Asset_Search”, “1-10”, “50”);
<!-- or more usefully -->
^International(“search title”, “Asset_Search”, <tmpl_var currentPage>, <tmpl_var total>);
```

Help

In addition to internationalization, WebGUI also provides an online help system for template variables, asset configuration, or anything else. This is done with an i18n-like module in the lib/WebGUI/Help folder (with the same name as your i18n namespace).

```
package WebGUI::Help::Asset_InfoCard;
use strict;
our $HELP = {
    'howToUse' => {
        title    => 'help howToUse title',
        body     => 'help howToUse body',
    },
};
1; # I can handle the truth
```

This help file seems a little abstract, there's no help to be found! What you do have, however, are keys to the i18n file so that Help can be internationalized as well.

First, set up your package to have the same name under WebGUI::Help as your i18n has under WebGUI::i18n::English. Then, make your $HELP structure, whose keys are references to the article. Inside the article reference, specify a title and body. The values of these two keys point to entries in the i18n file. This is all that is required to have a very basic help article. Once you add your keys to your i18n file, you can visit WebGUI's Help pane in the Admin Console and see your new article listed.

Template Variables

There are some help topics that are so common you have better ways to make help files for them. One of these topics is Template Variables.

By using the “variables” key in your article definition, you can specify a list of template variables available to a template, like so:

```
    'templateIdView' => {
        title     => 'help templateIdView title',
        body      => 'help templateIdView body',
        variables => [
        {
           name         => 'name',
           description  => 'helpvar name',
           required     => 0,
        },
        {
           name         => 'address',
           description  => 'helpvar address',
        },
     ],
  },
```

Here, a new help article is defined, templateIdView. You have the normal title and body keys, but you also have a new key: variables.

“variables” is an array reference of hash references that define each of the template variables available at that scope. For the InfoCard asset, you have at least two variables available “name” and “address”. The “name” key is the exact name of the template variable, what the template designer would use inside the <tmpl_var> tag. The “description” key is another i18n key for a description of the variable. Finally, the optional “required” key can be set to true if the variable absolutely must be inside the template for the asset to work.

If the variable is a loop, it can contain a second level of “variables”, like so:

```
     {
        name         => “card_loop”,
        description  => 'helpvar card_loop',
        variables    => [
           {
              name        => 'card',
              description => 'helpvar card',
           },
        ],
     },
```

These variables will show up with an extra level of indentation, to show they are available from inside the loop “card_loop.”

Related and ISA

A lot of assets are related to each other in some way. At the very least, they all inherit from the same base class, so share a portion of the same template variables. You can define these relationships in two ways: “related” and “isa”.

“isa” defines a standard parent-child relationship. By saying an article “isa” another article, you inherit all those template variables, and don't have to redefine them in your own article.

```
'templateIdView' => {
    title     => 'help templateIdView title',
    body      => 'help templateIdView body',
    isa       => [
        {
            namespace => 'Asset',
            tag       => 'asset template asset variables',
        },
    ],
    # variables
},
```

“isa” is an array reference of hash references, much like “variables.” Inside the hash reference is the “namespace” of the parent article, and the “tag” of the parent article, which is to say the key in the $HELP hashref. So, given the above isa, you could look in lib/WebGUI/Help/Asset.pm to find the “asset template asset variables” article that you're inheriting from.

“related” simply defines a relationship between two articles and provides a link to the related article. It's defined in the same way as “isa”, an array reference of hash references.

```
'templateIdView' => {
        title     => 'help templateIdView title',
        body      => 'help templateIdView body',
        related      => [
            {
                namespace => 'Asset',
                tag       => 'asset template asset variables',
            },
        ],
        # variables
    },
```

Now, the “asset template asset variables” article in the “Asset” namespace will show up under the “Related” section.

Writing SKU's

This chapter shows how to write SKU's, subclasses of WebGUI::Asset::Sku, which are assets that tie into the WebGUI Shop. Creating your own SKU's provides your customers a custom shopping experience.

WebGUI Shop comes with many SKU's that you can use as examples to guide you in writing your own SKU's. The Event Manager has four SKU's of its own: EMSTicket, EMSBadge, EMSRibbon, and EMSToken. In addition, there are the Product and Subscription SKU's, which can be used to generically sell any kind of products or recurring memberships. There are also some specialty SKU's, specifically the Donation and Flat Discount Coupon SKU's. It's wise to examine these SKU's before developing your own to get a feel for the effort involved, and to give you ideas on how to build your own.

API Highlights

The rich SKU API provides the power and flexibility needed to develop your own subclasses.

addToCart()

The most important method in all SKU's is the addToCart() method, because this is where the transaction begins. In some cases you may not need to modify the addToCart() method for your SKU, but in most cases you will probably either modify it or, at the very least, write a wrapper around it that calls it.

The main instance where you would want to modify the addToCart() method is if you have some book keeping functions to perform. For example, if this is a product with an inventory, you'll likely want to subtract the item out of inventory so that it can't be oversold.

The base addToCart() method accepts a hash reference of configuration properties and then calls the methods necessary to put this product into the cart. The configuration properties are used if you need to keep track of how the product is configured. For example, if it's a donation, you could put the donation amount in the properties. If it's a configurable product, like a computer, then you could store all of those configuration options in there.

Whenever the cart, or other parts of the shop, interact with this SKU, it will restore this SKU with these options using the applyOptions() and getOptions() methods.

The addToCart() method is generally never called by any WebGUI subsystems. It's there for you to write a wrapper around and call it yourself. Generally, you might create a www_addToCart() method that would be called by a user clicking on a button. How that happens is up to you.

Event Handlers

The SKU provides a series of event handlers which help manage the purchase process. There are some that are specific to the type of SKU that you're building. Others are useful in all SKU's: onCompletePurchase(), onRefund(), and onRemoveFromCart().

- onCompletePurchase() is called when a transaction has finalized and payment has been made. Use onCompletePurchase() to handle book keeping tasks like giving privileges, accounting for inventory, etc.
- onRefund() is called when a shop manager clicks the Refund button in the transaction manager. Use onRefund() to take away privileges, restock inventory that was previously purchased, etc.
- onRemoveFromCart() is called when an item is removed from the shopping cart before purchase. Use onRemoveFromCart() to restock items that you took out of inventory with the addToCart() method.

The Numbers

In a sale it all comes down to the numbers, so there are several methods that help you manage the numbers for each SKU.

- The getMaxAllowedInCart() method should be overridden in the event you're selling a unique item (for instance a ticket that has been customized to a user).
- The getPrice() method should be overridden to return the price of the SKU, and should calculate the price in the case of a configurable product.
- The getQuantityAvailable() method should be overridden if there are

a limited number of the SKU available for sale.

Display Helpers

There are a couple of methods that should be overridden to help visually integrate your SKU with the Shop.

- getConfiguredTitle() should return a descriptive title, such as “Red XL T-Shirt” rather than just “T-Shirt”.
- getThumbnailUrl() should return the URL to a thumbnail of a picture of the SKU.

SKU's as Products

The most common archetype of a SKU is a product: a physical good set up for sale. There is a generic Product SKU, that comes with WebGUI, that can handle the needs of most users. However, there are many scenarios in which using a SKU to do just the right job makes sense. That's where you come in to write a new product SKU.

Special Methods

There are a few special methods that you'll need to know when writing this type of SKU.

- The getWeight() method should be overridden to reflect the item's shipping weight.
- The isShippingRequired() method should be overridden to return 1 if the good is non-digital (a book rather than a PDF), or if the item will be picked up rather than shipped (like will-call tickets at a box office).
- The onAdjustQuantityInCart() method is an event handler, which is called when the user updates the quantity of an item in the cart. Use onAdjustQuanityInCart() to adjust inventory levels so that you don't over-sell an item.

Book Example

This shows how to write a SKU as a product archetype through the example of selling a book. Books have several properties that are unique to each book. A normal Product SKU could do the job, but not as well as a SKU written specifically for selling books. This example uses the special properties of ISBN, Author, and Edition.

As with any asset, start by creating the definition(). Add the author, ISBN, and edition properties, as well as the weight and price of the book.

```
sub definition {
	my $class = shift;
	my $session = shift;
	my $definition = shift;
	my %properties;
	tie %properties, 'Tie::IxHash';
	%properties = (
		isbn => {
			tab			=> "properties",
			fieldType		=> "text",
			defaultValue		=> undef,
			label			=> 'ISBN',
			hoverHelp		=> 'The International Standard Book Number.',
			},
		author => {
			tab			=> "properties",
			fieldType		=> "text",
			defaultValue		=> undef,
			label			=> 'Author',
			hoverHelp		=> 'Author\'s name or pen name.',
			},
		edition => {
			tab			=> "properties",
			fieldType		=> "text",
			defaultValue		=> 'First Edition',
			label			=> 'Edition',
			hoverHelp		=> 'The printing number or name of the book.',
			},
		price => {
			tab			=> "shop",
			fieldType		=> "float",
			defaultValue		=> 0,
			label			=> 'Price',
			hoverHelp		=> 'The amount this book sells for.',
			},
                weight => {
                        tab                     => "shop",
                        fieldType               => "float",
                        defaultValue            => 0,
                        label                   => 'Weight',
                        hoverHelp               => 'How much in lbs does this book weigh?',
                        },
        );
        push(@{$definition}, {
                assetName               => 'Book',
                icon                    => 'assets.gif',
                autoGenerateForms               => 1,
                tableName               => 'Book',
                className               => 'WebGUI::Asset::Sku::Book',
                properties              => \%properties
        });
        return $class->SUPER::definition($session, $definition);
}
```

Because you're building a product archetype, fill out the getPrice() and getWeight() methods. Note that the price formatted is returned using sprintf. This saves you from having to do it everywhere that you want to display the price.

```
sub getPrice {
        my $self = shift;
        return sprintf "%.2f", $self->get('price');
}

sub getWeight {
        my $self = shift;
        return $self->get('weight');
}
```

Then, define your view() method. Normally, you would want to template this, but to keep it simple this example has left out that step. Instead, all the properties are displayed using a standard HTML Form.

```
sub view {
        my $self = shift;
        my $f = WebGUI::HTMLForm->new($self->session, action=>$self->getUrl);
        $f->readOnly(
                label  => 'Title',
                value => $self->get('title'),
                );
        $f->readOnly(
                label  => 'Edition',
                value => $self->get('edition'),
                );
        $f->readOnly(
                label  => 'Description',
                value => $self->get('description'),
                );
        $f->readOnly(
                label  => 'Author',
                value => $self->get('author'),
                );
        $f->readOnly(
                label  => 'ISBN',
                value => $self->get('isbn'),
                );
        $f->readOnly(
                label  => 'Price',
                value => $self->getPrice,
                );
        $f->hidden(
                name => "func",
                value => "buy",
                );
        $f->submit( value => 'Add To Cart');
        return $f->print;
}
```

In view() you defined a reference to a www_buy() method, so you must also create that. This is the wrapper that will call addToCart() for you.

```
sub www_buy {
        my $self = shift;
        return $self->session->privilege->noAccess() unless ($self->canView);
        $self->addToCart;
        return $self->getParent->www_view;
}
```

Note that you want www_buy() to follow the same view privileges as all other assets. If you can't view it, you shouldn't be able to buy it. In this way you can make some books available to only certain groups of purchasers.

That gives you a final class that looks similar to the one below. This is also uploaded to the add-ons area of webgui.org, so you can work directly from this code if you plan on making something similar.

```
package WebGUI::Asset::Sku::Book;
use strict;
use Tie::IxHash;
use base 'WebGUI::Asset::Sku';
use WebGUI::Utility;
use WebGUI::HTMLForm;

=head1 NAME
Package WebGUI::Asset::Sku::Book
=head1 DESCRIPTION
This sku allows you to sell books on your site with complete information.
=head1 SYNOPSIS
use WebGUI::Asset::Sku::Book;

=head1 METHODS
These methods are available from this class:
=cut

#-------------------------------------------------------------------
=head2 definition ( session, definition )
Adds isbn, author, and edition fields.

=head3 session

=head3 definition

A hash reference passed in from a subclass definition.
=cut

sub definition {
	my $class = shift;
	my $session = shift;
	my $definition = shift;
	my %properties;
	tie %properties, 'Tie::IxHash';
```

```
%properties = (
        isbn => {
                tab                     => "properties",
                fieldType               => "text",
                defaultValue            => undef,
                label                   => 'ISBN',
                hoverHelp               => 'The International Standard Book Number.',
                },
        author => {
                tab                     => "properties",
                fieldType               => "text",
                defaultValue            => undef,
                label                   => 'Author',
                hoverHelp               => 'Author\'s name or pen name.',
                },
        edition => {
                tab                     => "properties",
                fieldType               => "text",
                defaultValue            => 'First Edition',
                label                   => 'Edition',
                hoverHelp               => 'The printing number or name of the book.',
                },
        price => {
                tab                     => "shop",
                fieldType               => "float",
                defaultValue            => 0,
                label                   => 'Price',
                hoverHelp               => 'The amount this book sells for.',
                },
        weight => {
                tab                     => "shop",
                fieldType               => "float",
                defaultValue            => 0,
                label                   => 'Weight',
                hoverHelp               => 'How much in lbs does this book weigh?',
                },
);
push(@{$definition}, {
        assetName                       => 'Book',
        icon                            => 'assets.gif',
        autoGenerateForms                       => 1,
        tableName                       => 'Book',
        className                       => 'WebGUI::Asset::Sku::Book',
                properties                      => \%properties
        });
        return $class->SUPER::definition($session, $definition);
}

#-------------------------------------------------------------------

=head2 getPrice ()

Returns the value of the price field formatted as currency.

=cut

sub getPrice {
        my $self = shift;
        return sprintf "%.2f", $self->get('price');
}

#-------------------------------------------------------------------

=head2 getWeight ()

Returns the value of the price field formatted as currency.

=cut

sub getWeight {
        my $self = shift;
        return $self->get('weight');
}

#-------------------------------------------------------------------

=head2 indexContent ( )

Making private. See WebGUI::Asset::indexContent() for additonal details.

=cut

sub indexContent {
        my $self = shift;
        my $indexer = $self->SUPER::indexContent;
        $indexer->addKeywords($self->get('author'), $self->get('isbn'), $self->get('edition'));
```

```
}

#-------------------------------------------------------------------
=head2 view ( )

method called by the container www_view method.

=cut

sub view {
	my $self = shift;
	my $f = WebGUI::HTMLForm->new($self->session, action=>$self->getUrl);
	$f->readOnly(
		label   => 'Title',
		value  => $self->get('title'),
		);
	$f->readOnly(
		label   => 'Edition',
		value  => $self->get('edition'),
		);
	$f->readOnly(
		label   => 'Description',
		value  => $self->get('description'),
		);
	$f->readOnly(
		label   => 'Author',
		value  => $self->get('author'),
		);
	$f->readOnly(
		label   => 'ISBN',
		value  => $self->get('isbn'),
		);
	$f->readOnly(
		label   => 'Price',
		value  => $self->getPrice,
		);
	$f->hidden(
		name => "func",
		value  => "buy",
		);
	$f->submit( value => 'Add To Cart');
	return $f->print;
}
```

```
#-------------------------------------------------------------------

=head2 www_buy ()

Adds the book to the cart.

=cut

sub www_buy {
	my $self = shift;
	return $self->session->privilege->noAccess() unless ($self->canView);
	$self->addToCart;
	return $self->getParent->www_view;
}

1;
```

Recurring SKU's

Recurring SKU's are often thought of as subscriptions or memberships. You pay every so often to see a support board, to read some articles about your favorite sports team, or to pay your dues to a club. In many of these cases the Subscription SKU that comes with WebGUI will serve the purpose perfectly. However, you may want to collect some data along with the subscription, so you'd create your own recurring SKU.

Special Methods

There are a few special methods you need to be aware of when writing a recurring SKU archetype.

- The getRecurInterval() method should be overridden so that the Shop knows how often to process the recurring transaction.
- The isRecurring() method should be overridden to return 1 so that the Shop knows this is a recurring SKU.
- The onCancelRecurring() method is an event handler, which is called when a user or administrator cancels a recurring transaction. Use this to make privilege adjustments, send out emails, etc. However, be careful not to take away a user's privileges too early. This cancels the next transaction, but if you're working with a subscription, and

that subscription has not already expired, then the user needs to maintain its current privileges, and lose them when the next transaction fires.

Association Dues Example

In clubs and associations, it's often necessary to collect dues and members' information. You may want to collect these simultaneously the first time, and then allow your members to update their data periodically thereafter. A great way to achieve this is to display a form of the data you want to capture with the add to cart button. For this example, data is collected and placed in the user's profile. That way the user can use the profile system to update the data going forward.

Start with the definition() method, which defines the memberGroupId (the group that holds your members) and the price fields.

```
sub definition {
	my $class = shift;
	my $session = shift;
	my $definition = shift;
	my %properties;
	tie %properties, 'Tie::IxHash';
	%properties = (
		memberGroupId => {
			tab			=> "properties",
			fieldType		=> "group",
			defaultValue		=> '3',
			label			=> 'Member Group',
			hoverHelp		=> 'The group holding your membership.',
			},
		price => {
			tab			=> "shop",
			fieldType		=> "float",
			defaultValue		=> 0,
			label			=> 'Price',
			hoverHelp		=> 'The annual fee for this membership.',
			},
	);
	push(@{$definition}, {
		assetName			=> 'Member Dues',
		icon				=> 'assets.gif',
		autoGenerateForms			=> 1,
                tableName                       => 'MemberDues',
                className                       => 'WebGUI::Asset::Sku::MemberDues',
                properties                      => \%properties
        });
        return $class->SUPER::definition($session, $definition);
}
```

Then, you have to publish your price.

```
sub getPrice {
        my $self = shift;
        return sprintf "%.2f", $self->get('price');
}
```

Because this is a recurring SKU, you have to override the appropriate method to let the Shop know that. By defining isRecurring to return 1, getMaxAllowedInCart() will automatically also return 1. In this case, you're building a SKU that recurs annually, so you can hard code getRecurInterval(). In some cases you may want to make this a select box property so that the user can select the recur interval.

```
sub getRecurInterval {
    return 'Yearly';
}

sub isRecurring {
        return 1;
}
```

You need to set up the privileges on each recurrence, but you only want to update the membership data on the first recurrence. Do that with the onCompletePurchase() method.

```
sub onCompletePurchase {
        my ($self, $item) = @_;
        my $transaction = $item->transaction;
        my $group = WebGUI::Group->new($self->session, $self->get('memberGroupId'));
        $group->addUsers([$transaction->get('userId')],60*60*24*365);
        if ($transaction->isFirst) { # this is the first instance of this transaction
                my $user = WebGUI::User->new($self->session, $transaction->get('userId'));
                my $options = $self->getOptions;
```

```
		$user->profileField('email', $options->{email});
		$user->profileField('firstName', $options->{firstName});
		$user->profileField('lastName', $options->{lastName});
		$user->profileField('workPhone', $options->{workPhone});
	}
}
```

Then, you can build out the view() method with your registration form.

```
sub view {
	my $self = shift;
	my $user = $self->session->user;
	my $f = WebGUI::HTMLForm->new($self->session, action=>$self->getUrl);
	$f->readOnly(
		label	=> 'Price',
		value	=> $self->getPrice,
		);
	$f->hidden(
		name => "func",
		value => "buy",
		);
	$f->text(
		name		=> 'firstName',
		defaultValue	=> $user->profileField('firstName'),
		label		=> 'First Name',
		);
	$f->text(
		name		=> 'lastName',
		defaultValue	=> $user->profileField('lastName'),
		label		=> 'Last Name',
		);
	$f->email(
		name		=> 'email',
		defaultValue	=> $user->profileField('email'),
		label		=> 'Email Address',
		);
	$f->phone(
		name		=> 'workPhone',
		defaultValue	=> $user->profileField('workPhone'),
		label		=> 'Telephone Number',
		);
	$f->submit( value => 'Add To Cart');
	my $output = q{
		<h3>}.$self->getTitle.q{</h3>
		<p>}.$self->get('description').q{</p>
		<p>}.$f->print.q{</p>
	};
	return $output;
}
```

Just like in the Book product, you have to create the www_buy() method, but this time you have some configuration options to store with the item. Note the use of that in the addToCart() method.

```
sub www_buy {
	my $self = shift;
	return $self->session->privilege->noAccess() unless ($self->canView);
	my $form = $self->session->form;
	$self->addToCart({
		workPhone  => $form->get('workPhone','phone'),
		email      => $form->get('email','email'),
		firstName  => $form->get('firstName','text'),
		lastName   => $form->get('lastName','text'),
		});
	return $self->getParent->www_view;
}
```

All of the methods together give you an asset that looks like this:

```
package WebGUI::Asset::Sku::MemberDues;

use strict;
use Tie::IxHash;
use base 'WebGUI::Asset::Sku';
use WebGUI::Utility;
use WebGUI::HTMLForm;
use WebGUI::Group;


=head1 NAME

Package WebGUI::Asset::Sku::MemberDues

=head1 DESCRIPTION
```

```
This sku allows you to register new members to your organization.

=head1 SYNOPSIS

use WebGUI::Asset::Sku::MemberDues;


=head1 METHODS

These methods are available from this class:

=cut

#-------------------------------------------------------------------

=head2 definition ( session, definition )

Adds  memberGroupId and price fields.

=head3 session

=head3 definition

A hash reference passed in from a subclass definition.

=cut

sub definition {
	my $class = shift;
	my $session = shift;
	my $definition = shift;
	my %properties;
	tie %properties, 'Tie::IxHash';
	%properties = (
		memberGroupId => {
			tab			=> "properties",
			fieldType		=> "group",
			defaultValue		=> '3',
			label			=> 'Member Group',
			hoverHelp		=> 'The group holding your membership.',
			},
		price => {
```

```
                        tab                    => "shop",
                        fieldType              => "float",
                        defaultValue           => 0,
                        label                  => 'Price',
                        hoverHelp              => 'The annual fee for this membership.',
                        },
        );
        push(@{$definition}, {
                assetName                      => 'Member Dues',
                icon                           => 'assets.gif',
                autoGenerateForms                      => 1,
                tableName                      => 'MemberDues',
                className                      => 'WebGUI::Asset::Sku::MemberDues',
                properties                     => \%properties
        });
        return $class->SUPER::definition($session, $definition);
}

#-------------------------------------------------------------------

=head2 getPrice ()

Returns the value of the price field formatted as currency.

=cut

sub getPrice {
        my $self = shift;
        return sprintf "%.2f", $self->get('price');
}

#-------------------------------------------------------------------

=head2 getRecurInterval ( )

Returns 'Yearly'.

=cut

sub getRecurInterval {
    return 'Yearly';
}
```

```
#-------------------------------------------------------------------

=head2 isRecurring ( )

Returns 1.

=cut

sub isRecurring {
	return 1;
}

#-------------------------------------------------------------------

=head2 onCompletePurchase ( item )

Adds the user to the member group and if it's the first time, applies the fields to the profile.

=cut

sub onCompletePurchase {
	my ($self, $item) = @_;
	my $transaction = $item->transaction;
	my $group = WebGUI::Group->new($self->session, $self->get('memberGroupId'));
	$group->addUsers([$transaction->get('userId')],60*60*24*365);
	if ($transaction->isFirst) { # this is the first instance of this transaction
		my $user = WebGUI::User->new($self->session, $transaction->get('userId'));
		my $options = $self->getOptions;
		$user->profileField('email', $options->{email});
		$user->profileField('firstName', $options->{firstName});
		$user->profileField('lastName', $options->{lastName});
		$user->profileField('workPhone', $options->{workPhone});
	}
}

#-------------------------------------------------------------------
=head2 view ( )

method called by the container www_view method.

=cut

sub view {
	my $self = shift;
	my $user = $self->session->user;
	my $f = WebGUI::HTMLForm->new($self->session, action=>$self->getUrl);
	$f->readOnly(
		label  => 'Price',
		value  => $self->getPrice,
		);
	$f->hidden(
		name => "func",
		value => "buy",
		);
	$f->text(
		name		=> 'firstName',
		defaultValue	=> $user->profileField('firstName'),
		label		=> 'First Name',
		);
	$f->text(
		name		=> 'lastName',
		defaultValue	=> $user->profileField('lastName'),
		label		=> 'Last Name',
		);
	$f->email(
		name		=> 'email',
		defaultValue	=> $user->profileField('email'),
		label		=> 'Email Address',
		);
	$f->phone(
		name		=> 'workPhone',
		defaultValue	=> $user->profileField('workPhone'),
		label		=> 'Telephone Number',
		);
	$f->submit( value => 'Add To Cart');
	my $output = q{
		<h3>}.$self->getTitle.q{</h3>
		<p>}.$self->get('description').q{</p>
		<p>}.$f->print.q{</p>
	};
	return $output;
}

#-------------------------------------------------------------------

=head2 www_buy ()
```

```
Adds the book to the cart.

=cut

sub www_buy {
	my $self = shift;
	return $self->session->privilege->noAccess() unless ($self->canView);
	my $form = $self->session->form;
	$self->addToCart({
		workPhone  => $form->get('workPhone','phone'),
		email      => $form->get('email','email'),
		firstName  => $form->get('firstName','text'),
		lastName   => $form->get('lastName','text'),
		});
	return $self->getParent->www_view;
}

1;
```

SKU's as Coupons

Another powerful aspect of SKU's is that they know what else is in the cart and can adjust the price accordingly. This allows SKU's to be used as coupons. For example, you might detect that the user has placed an order for 10 packs of tube socks in the cart. You just happen to be running a special on tube socks, which gives the user a 20% discount if s/he buys 5 or more packages. Your coupon can look at the cart, determine if the user has met the criteria, and then return a negative price, thus creating a discount.

Special Methods

There are a couple of methods you should be aware of when creating a SKU as coupon archetype.

- The getCart() method gives you a reference to the shopping cart (WebGUI::Shop::Cart) so that you can see what else is in the cart, to make your judgments about discounts to give.

- You also need to override isCoupon() to return a 1, because some coupons may want to know if the user has other coupons in the cart, and only allow one coupon per cart.

Member Discount Example

Let's say that you have an online shop, which sells training materials for realtors, and that you also run a realtor association from your web site. You may want to give a discount to all the realtors that also belong to your association. So, create a coupon that your members can add to their carts that gives them their association discount.

Similar to the Member Dues example, start out your definition() method with a memberGroupId property. This time, instead of price, you have a discount field.

```
sub definition {
        my $class = shift;
        my $session = shift;
        my $definition = shift;
        my %properties;
        tie %properties, 'Tie::IxHash';
        %properties = (
                memberGroupId => {
                        tab             => "properties",
                        fieldType       => "group",
                        defaultValue    => '3',
                        label           => 'Member Group',
                        hoverHelp       => 'The group holding your membership.',
                        },
                discount => {
                        tab             => "shop",
                        fieldType       => "integer",
                        defaultValue    => 10,
                        label           => 'Percentage Discount',
                        hoverHelp       => 'The amount of discount in percent.',
                        },
        );
        push(@{$definition}, {
                assetName               => 'Member Discount',
                icon                    => 'assets.gif',
                autoGenerateForms               => 1,
                tableName               => 'MemberDiscount',
                className               => 'WebGUI::Asset::Sku::MemberDiscount',
                properties              => \%properties
        });
        return $class->SUPER::definition($session, $definition);
}
```

With the coupon archetype most of the work happens in the getPrice() method. Only give the discount if the user is a member of your group. Also note that when checking the items in the cart you skip yourself. If you didn't do this you would create an infinite loop.

```
sub getPrice {
	my $self = shift;
	my $discount = 0;
	if ($self->session->user->isInGroup($self->get('memberGroupId'))) {
		foreach my $item (@{$self->getCart->getItems}) {
			next if ($item->get('assetId') eq $self->getId); # prevent infinite loop
			$discount += $item->sku->getPrice * $item->get('quantity') * $self-
>get('discount') * -1 / 100;
		}
	}
	return sprintf "%.2f", $discount;
}
```

Because this is a coupon you have to tell the Shop that.

```
sub isCoupon {
	return 1;
}
```

And your view() method looks very simple this time.

```
sub view {
	my $self = shift;
	my $f = WebGUI::HTMLForm->new($self->session, action=>$self->getUrl);
	$f->hidden(
		name => "func",
		value  => "addToCart",
		);
	$f->submit( value => 'Add To Cart');
	my $output = q{
		<h3>}.$self->getTitle.q{</h3>
		<p>}.$self->get('description').q{</p>
		<p>}.$f->print.q{</p>
	};
	return $output;
}
```

But the www_addToCart() method looks pretty similar to the www_buy() method in the Book example.

```
sub www_addToCart {
	my $self = shift;
	return $self->session->privilege->noAccess() unless ($self->canView);
	$self->addToCart;
	return $self->getParent->www_view;
}
```

If you put all that together, you get an asset that looks similar to this:

```
package WebGUI::Asset::Sku::MemberDiscount;

use strict;
use Tie::IxHash;
use base 'WebGUI::Asset::Sku';
use WebGUI::Utility;
use WebGUI::HTMLForm;


=head1 NAME

Package WebGUI::Asset::Sku::MemberDiscount

=head1 DESCRIPTION

This sku gives members of a specific group a discount.

=head1 SYNOPSIS

use WebGUI::Asset::Sku::MemberDiscount;


=head1 METHODS

These methods are available from this class:

=cut

#-------------------------------------------------------------------
```

```
=head2 definition ( session, definition )

Adds memberGroupId and discount fields.

=head3 session

=head3 definition

A hash reference passed in from a subclass definition.

=cut

sub definition {
        my $class = shift;
        my $session = shift;
        my $definition = shift;
        my %properties;
        tie %properties, 'Tie::IxHash';
        %properties = (
                memberGroupId => {
                        tab             => "properties",
                        fieldType       => "group",
                        defaultValue    => '3',
                        label           => 'Member Group',
                        hoverHelp       => 'The group holding your membership.',
                        },
                discount => {
                        tab             => "shop",
                        fieldType       => "integer",
                        defaultValue    => 10,
                        label           => 'Percentage Discount',
                        hoverHelp       => 'The amount of discount in percent.',
                        },
        );
        push(@{$definition}, {
                assetName               => 'Member Discount',
                icon                    => 'assets.gif',
                autoGenerateForms               => 1,
                tableName               => 'MemberDiscount',
                className               => 'WebGUI::Asset::Sku::MemberDiscount',
                properties              => \%properties
        });
        return $class->SUPER::definition($session, $definition);
}

#-------------------------------------------------------------------

=head2 getPrice ()

Returns the discount formatted as currency.

=cut

sub getPrice {
	my $self = shift;
	my $discount = 0;
	if ($self->session->user->isInGroup($self->get('memberGroupId'))) {
		foreach my $item (@{$self->getCart->getItems}) {
			next if ($item->get('assetId') eq $self->getId); # prevent infinite loop
			$discount += $item->sku->getPrice * $item->get('quantity') * $self-
>get('discount') * -1 / 100;
		}
	}
	return sprintf "%.2f", $discount;
}

#-------------------------------------------------------------------

=head2 isCoupon ()

Returns 1.

=cut

sub isCoupon {
	return 1;
}

#-------------------------------------------------------------------

=head2 view ( )

method called by the container www_view method.

=cut
```

```
sub view {
        my $self = shift;
        my $f = WebGUI::HTMLForm->new($self->session, action=>$self->getUrl);
        $f->hidden(
                name => "func",
                value  => "addToCart",
                );
        $f->submit( value => 'Add To Cart');
        my $output = q{
                <h3>}.$self->getTitle.q{</h3>
                <p>}.$self->get('description').q{</p>
                <p>}.$f->print.q{</p>
        };
        return $output;
}

#-------------------------------------------------------------------

=head2 www_addToCart ()

Adds the coupon to the cart.

=cut

sub www_addToCart {
        my $self = shift;
        return $self->session->privilege->noAccess() unless ($self->canView);
        $self->addToCart;
        return $self->getParent->www_view;
}

1;
```

Writing Payment Drivers

Payment drivers are plugins to WebGUI Shop that accept payment for an order. They may tie in to a traditional credit card payment gateway, or an e-check service, or to a payment service like Pay Pal or Google Checkout, or even to a custom billing system. No matter what they connect to, the idea is the same: verify and accept payment for an order purchased through WebGUI shop.

API Highlights

WebGUI comes with two payment drivers. The first, Cash, is good for point of sale type orders. The other is iTransact, which is a typical traditional credit card payment gateway service. Both are great reference material that you should have a look at and take the time to understand before you start writing your own payment driver. When you are ready to write your own payment driver, there are several parts of the API you should be aware of, and that's what this section is about.

The Master Classes

Before you start writing your own payment driver subclass, check out WebGUI::Shop::Pay and WebGUI::Shop::PayDriver. Pay is the management interface for all PayDrivers. PayDriver is the master class that you'll be subclassing to create your own.

Pay hands off web requests to the PayDrivers. It does this through its www_do() method. Along the URL it looks like:

/home?shop=pay;method=do;driverId=XXXXX;do=**someMethod**

Note the bolded "someMethod" part of the URL. That refers to a www_**someMethod**() method in your PayDriver subclass. This allows you to build web accessible content through your payment methods, and use that to integrate your payment module with any payment gateway you want to over HTTP.

PayDriver Basics

There are a few methods you're going to need to know in order to write your own payment driver.

The first is the definition() method. If you've written an asset or workflow activity then you'll be familiar with how this works. In definition() you specify the properties of the driver and its human readable name.

You'll need to override the getButton() method and return an HTML button that the user will click on the select this payment method. This is generally the entrance point into a series of www_ methods that you'll create. However, for external payment systems like PayPal, this may redirect the user to the external system.

You'll then need to override the processPayment() method, which will be called by the processTransaction() method. The processPayment() method should actually perform the payment request to the payment gateway. This needs to be an atomic (pass/fail) type of transaction; either the whole thing succeeds or the whole thing fails.

Recurring Methods

If your payment gateway can be used to handle recurring payments (most can't), then there are a few additional methods you'll need to be aware of.

You'll need to override the cancelRecurringPayment() method, which should make a call to the payment gateway requesting that the recurring transaction be terminated. Like processPayment() this request must be atomic.

You'll also need to override the handlesRecurring() method and make it return 1. This way WebGUI Shop knows this driver is capable of handling recurring payments.

Karma Payment Example

This example creates a payment gateway that uses Karma, instead of money, to pay for goods. This is not only an interesting concept, but it requires nothing other than WebGUI to use it. This means you'll be able to test it without getting involved with some external payment gateway service, which can be a daunting task if you haven't done it before.

First, create your definition() method, where you'll define a special property used as a currency conversion ratio. In other words, how many points of karma does it take to purchase an item worth $1.

```
sub definition {
   my $class       = shift;
   my $session     = shift;
   my $definition  = shift;

   tie my %fields, 'Tie::IxHash';

   %fields = (
      conversionRatio => {
         fieldType       => 'integer',
         label           => 'Conversion Ratio',
         hoverHelp       => 'The amount of karma it takes to equal 1 of the shop\'s currency
(dollars, yen, euros, etc)',
         defaultValue    => 500,
         subtext         => 'karma : 1 currency'
      },
      );

   push @{ $definition }, {
      name        => 'Karma',
      properties  => \%fields,
   };

   return $class->SUPER::definition($session, $definition);
}
```

Now, define the getButton() method. Note the use of getDoFormTags(), which is just a helper that adds the shop, method, do, and payId variables to the form as hidden input tags. Redirect to the www_pay() method here.

```
sub getButton {
   my $self     = shift;
   my $session = $self->session;
   my $payForm = WebGUI::Form::formHeader($session)
      . $self->getDoFormTags('pay')
      . WebGUI::Form::submit($session, {value => 'Karma' })
      . WebGUI::Form::formFooter($session);
```

```
    return $payForm;
}
```

Now you can write the payment processing method, which checks to see that the user has enough karma, then subtracts it for the payment.

```
sub processPayment {
    my ($self, $transaction) = @_;
    my $user = WebGUI::User->new($self->session, $transaction->get('userId'));
    if ($user->karma / $self->get('conversionRatio') > $transaction->get('amount')) {
        $user->karma(
          $transaction->get('amount') * -1,
          'Shop',
          'Purchase of Order #'.$transaction->get('orderNumber')
          );
        return (1,          # success
                undef,      # transaction code
                1,          # status code
                'Success'); # status message
    }
    return (0,                      # success
            undef,                  # transaction code
            0,                      # status code
            'Insufficient Karma');  # status message
}
```

Finish up the driver with the method that processes the payment and displays the thank you page.

```
sub www_pay {
    my $self    = shift;
    my $cart    = $self->getCart;

    # Make sure you can checkout the cart
    return "" unless $cart->readyForCheckout;

    # Complete the transaction
    my $transaction = $self->processTransaction( );
    return $transaction->thankYou();
}
```

The finished class will look something like the following:

```
package WebGUI::Shop::PayDriver::Karma;

use strict;

use WebGUI::Shop::PayDriver;
use WebGUI::Exception;

use base qw/WebGUI::Shop::PayDriver/;

#-------------------------------------------------------------------

sub definition {
    my $class       = shift;
    my $session     = shift;
    my $definition  = shift;

    tie my %fields, 'Tie::IxHash';

    %fields = (
        conversionRatio => {
            fieldType       => 'integer',
            label           => 'Conversion Ratio',
            hoverHelp       => 'The amount of karma it takes to equal 1 of the shop\'s currency
(dollars, yen, euros, etc)',
            defaultValue    => 500,
            subtext         => 'karma : 1 currency'
        },
        );

    push @{ $definition }, {
        name        => 'Karma',
        properties  => \%fields,
    };

    return $class->SUPER::definition($session, $definition);
}

#-------------------------------------------------------------------

sub getButton {
    my $self    = shift;
    my $session = $self->session;
    my $payForm = WebGUI::Form::formHeader($session)
        . $self->getDoFormTags('pay')
        . WebGUI::Form::submit($session, {value => 'Karma' })
        . WebGUI::Form::formFooter($session);

    return $payForm;
}

#-------------------------------------------------------------------

sub processPayment {
    my ($self, $transaction) = @_;
    my $user = WebGUI::User->new($self->session, $transaction->get('userId'));
    if ($user->karma / $self->get('conversionRatio') > $transaction->get('amount')) {
        $user->karma(
            $transaction->get('amount') * -1,
            'Shop',
            'Purchase of Order #'.$transaction->get('orderNumber')
            );
        return (1,          # success
                undef,      # transaction code
                1,          # status code
                'Success'); # status message
    }
    return (0,                      # success
            undef,                  # transaction code
            0,                      # status code
            'Insufficient Karma');  # status message
}

#-------------------------------------------------------------------

sub www_pay {
    my $self    = shift;
    my $cart    = $self->getCart;

    # Make sure you can checkout the cart
    return "" unless $cart->readyForCheckout;

    # Complete the transaction
    my $transaction = $self->processTransaction( );
    return $transaction->thankYou();
```

```
}



1;
```

Writing Shipping Drivers

Shipping drivers allow you to calculate and charge for shipping costs, and in some cases tie into third-party shipping providers such as UPS or FedEx.

API Highlights

WebGUI comes with one shipping driver, FlatRate, which can calculate most basic shipping costs, but does not tie into any specific shipping provider. It is great reference material that you should have a look at and take the time to understand before you start writing your own shipping driver. When you are ready to write your own shipping driver, there are several parts of the API you should be aware of, and that's what this section is about.

The Master Classes

Just like with writing payment drivers, before you start writing your own shipping driver subclass, check out WebGUI::Shop::Ship and WebGUI::Shop::ShipDriver. Ship is the management interface for all ShipDrivers. ShipDriver is the master class that you'll be subclassing to create your own.

Ship hands off web requests to the ShipDrivers. It does this through its www_do() method. Along the URL it looks like:

/home?shop=ship;method=do;driverId=XXXXX;do=**someMethod**

Note the bolded "someMethod" part of the URL. That refers to a www_**someMethod**() method in your ShipDriver subclass. Generally speaking you won't need this ability, but it's there if you do.

The Basics

There are really only two methods you should need to fill out in your shipping drivers, unless you do shipment tracking integration with a third party service. They are calculate() and definition().

- The calculate method is the true heart of a ShipDriver. This method gets a reference to the cart, and from that calculates the shipping cost on the products in the cart.

- The definition() method works just like the definition method in PayDrivers and is very similar to the definition() methods in Assets and Workflow Activities.

Free Shipping on Big Orders Example

Most shops would love it if their average order price were greater than $100. You can actually encourage that behavior using a custom shipping driver. Let's build a driver where the users will pay a base shipping fee of X and Y per item, unless the transaction is worth Z. For example, they will pay a base fee of $5 + $2 per item unless their transaction is > $100.

First, build the definition() method so you can define X, Y, and Z.

```
sub definition {
    my $class      = shift;
    my $session    = shift;
    WebGUI::Error::InvalidParam->throw(error => q{Must provide a session variable})
        unless ref $session eq 'WebGUI::Session';
    my $definition = shift || [];
    tie my %fields, 'Tie::IxHash';
    %fields = (
        baseFee => {
            fieldType    => 'float',
            label        => 'Base Fee',
            hoverHelp    => 'The amount of shipping applied to all orders.',
            defaultValue => 5.00,
        },
        perItemFee => {
            fieldType    => 'float',
            label        => 'Per Item Fee',
            hoverHelp    => 'The additional shipping charge per item.',
            defaultValue => 2.00,
        },
        freeThreshold => {
            fieldType    => 'float',
            label        => 'Free Threshold',
            hoverHelp    => 'The total amount of the order before shipping is free.',
            defaultValue => 100,
        },
    );
    my %properties = (
        name        => 'Free Shipping On Big Orders',
        properties  => \%fields,
    );
    push @{ $definition }, \%properties;
    return $class->SUPER::definition($session, $definition);
}
```

Now that you have the properties to work with, you can define your calculate() method.

```
sub calculate {
    my ($self, $cart) = @_;
          my $quantityOfShippableItems = 0;
          foreach my $item (@{$cart->getItems}) {
                    my $sku = $item->getSku;
                    if ($sku->isShippingRequired) {
                              $quantityOfShippableItems += $item->get('quantity');
                    }
          }
          my $cost = 0;
          if ($quantityOfShippableItems > 0) {
                    $cost = $self->get('baseFee') + ($quantityOfShippableItems * $self-
>get('perItemFee'));
          }
          if ($cost > $self->get('freeThreshold')) {
                    return '0.00';
          }
    return sprintf "%.2f", $cost;
}
```

The finished module should then look something like this:

```
package WebGUI::Shop::ShipDriver::FreeShippingOnBigOrders;

use strict;
use base qw/WebGUI::Shop::ShipDriver/;
use WebGUI::Exception;

=head1 NAME

```

```
Package WebGUI::Shop::ShipDriver::FreeShippingOnBigOrders

=head1 DESCRIPTION

This driver allows free shipping on big orders.

=head1 SYNOPSIS

=head1 METHODS

See the master class, WebGUI::Shop::ShipDriver for information about
base methods.  These methods are customized in this class:

=cut

#-------------------------------------------------------------------

=head2 calculate ( $cart )

Returns a shipping price. Calculates the shipping price using the following formula:

    baseFee + (perItemFee * item quantity) || 0 if (cart > freeThreshold)

=head3 $cart

A WebGUI::Shop::Cart object.  The contents of the cart are analyzed to calculate the
shipping costs.  If no items in the cart require shipping, then no shipping
costs are assessed.

=cut

sub calculate {
    my ($self, $cart) = @_;
        my $quantityOfShippableItems = 0;
        foreach my $item (@{$cart->getItems}) {
                my $sku = $item->getSku;
                if ($sku->isShippingRequired) {
                        $quantityOfShippableItems += $item->get('quantity');
                }
        }
        my $cost = 0;
        if ($quantityOfShippableItems > 0) {
                $cost = $self->get('baseFee') + ($quantityOfShippableItems * $self-
>get('perItemFee'));
        }
        if ($cost > $self->get('freeThreshold')) {
                return '0.00';
        }
    return sprintf "%.2f", $cost;
}

#-------------------------------------------------------------------

=head2 definition ( $session )

Add baseFee, perItemFee, and freeThreshold properties.

=cut

sub definition {
    my $class      = shift;
    my $session    = shift;
    WebGUI::Error::InvalidParam->throw(error => q{Must provide a session variable})
        unless ref $session eq 'WebGUI::Session';
    my $definition = shift || [];
    tie my %fields, 'Tie::IxHash';
    %fields = (
        baseFee => {
            fieldType    => 'float',
            label        => 'Base Fee',
            hoverHelp    => 'The amount of shipping applied to all orders.',
            defaultValue => 5.00,
        },
        perItemFee => {
            fieldType    => 'float',
            label        => 'Per Item Fee',
            hoverHelp    => 'The additional shipping charge per item.',
            defaultValue => 2.00,
        },
        freeThreshold => {
            fieldType    => 'float',
            label        => 'Free Threshold',
            hoverHelp    => 'The total amount of the order before shipping is free.',
            defaultValue => 100,
        },
    );
    my %properties = (
        name        => 'Free Shipping On Big Orders',
        properties  => \%fields,
    );
    push @{ $definition }, \%properties;
    return $class->SUPER::definition($session, $definition);
}

1;
```

Appendix 1: Advanced Install Scripts for Assets

As mentioned in the chapter on writing wobjects, there is an additional step you should add to your install methods that is not included in the _NewWobject.skeleton class, which is importing all of the default templates for your asset. This section details one way to do this which allows your templates to not only be automatically imported, but also updated without the need for manually copying and pasting, or committing.

The first step is to define a metadata scheme for your templates. By adding a few properties to the beginning of each template you can tell your code how it should be imported. In the example, each metadata property is prefaced with a pound sign (#). Your templates will look something like this:

```
#title=Default New Asset Template
#namespace=NewAsset/view
#assetId=NewAsset00000000000001

<tmpl_if session.var.adminOn><p><tmpl_var controls></p></tmpl_if>
<tmpl_if displayTitle><h2><tmpl_var title></h2></tmpl_if>
<tmpl_if message>
  <div class="error"><tmpl_var message></div>
</tmpl_if>

~~~

<style>
</style>
```

Store all such templates in a folder with .tmpl extensions.

```
/data/WebGUI/docs/NewAsset/templates/view.tmpl
```

Now, add some code which reads the directory and parses the template looking for the title, namespace, and assetId of your templates, as well as the body and head tags.

Start by exporting a new method:

```
use base 'Exporter';
our @EXPORT = qw(install uninstall upgrade);
use WebGUI::Session;
```

Now, update the install method to additionally install your templates.

```
sub install {
        my $config      = $ARGV[0];
        my $home        = $ARGV[1] || "/data/WebGUI";
        my $templateDir = $ARGV[2] || "$home/docs/NewAsset/templates";
        unless ($home && $config) {
           die "usage: perl -MWebGUI::Asset::Wobject::NewAsset -e install
www.example.com.conf\n";
        }

        #Open the session
        print "Installing asset.\n";
        my $session = WebGUI::Session->open($home, $config);
           my $db         = $session->db;

           #Change the user session to the admin
        $session->user({ userId=>3 });

        #Add the asset to the config file
        $session->config->addToArray("assets","WebGUI::Asset::Wobject::NewAsset");

        # Create asset table
           $db->write("create table if not exists NewAsset (
                assetId varchar(22) binary not null,
                revisionDate bigint not null,
                primary key (assetId, revisionDate)
        )");

           # Create a folder asset to store the default template
        my $importNode = WebGUI::Asset->getImportNode($session);
        my $newFolder  = $importNode->addChild({
                className   =>"WebGUI::Asset::Wobject::Folder",
                title          => "New Asset Folder",
                menuTitle    => "New Asset Folder",
                url             => "new_asset_folder",
                groupIdView =>"3"
        },"NewAssetFolder00000001");

        #Install the default templates
        importTemplates($session,$templateDir,$newFolder);

        #Commit the version tag containing the folder and all the templates
        my $tag = WebGUI::VersionTag->new($session, WebGUI::VersionTag-
>getWorking($session)->getId);
           if (defined $tag) {
           print "Committing tag\n";
           $tag->set({comments=>"Folder and Asset Templates created by Asset Install
Process"});
            $tag->requestCommit;
        }

        #Close the session
        $session->var->end;
        $session->close;
        print "Done. Please restart Apache.\n";
}
```

Note that another property was added that can be passed in, which is the location of the template directory. A default location has also been created for it. Additionally, code is added in the install portion which creates a new folder for which you've hardcoded an assetId. It is hard coded so you can later locate it to add, update, or remove templates.

Finally, a new method is added called importTemplates, which is used to do the template import. Once the import is complete, the folder and all the templates are committed.

The next thing to do is to update the uninstall method:

```
sub uninstall {
        my $config = $ARGV[0];
        my $home   = $ARGV[1] || "/data/WebGUI";
        unless ($home && $config) {
              die "usage: perl -MWebGUI::Asset::Wobject::NewAsset -e uninstall
www.example.com.conf\n" ;
        }

        #Open the session
        print "Uninstalling asset.\n";
        my $session = WebGUI::Session->open($home, $config);
        my $db      = $session->db;

        #Change the user session to the admin
        $session->user({userId=>3});
        #Add the asset to the config file
        $session->config-
>deleteFromArray("assets","WebGUI::Asset::Wobject::NewAsset");

        #Remove the assets from the site
        my $rs = $db->read("select assetId from asset where
className='WebGUI::Asset::Wobject::NewAsset'");
        while (my ($id) = $rs->array) {
             my $asset = WebGUI::Asset->new($session, $id,
"WebGUI::Asset::Wobject::NewAsset");
             $asset->purge if defined $asset;
        }

        #Remove the templates and the template folder
        my @templateIds = $db->buildArray(q|
                select distinct assetId from template where namespace in ('NewAsset/view')
            |);
        push(@templateIds,"NewAssetFolder00000001");
        foreach my $templateId (@templateIds) {
             my $asset = WebGUI::Asset->new($session, $templateId,
"WebGUI::Asset::Wobject::NewAsset");
             $asset->purge if defined $asset;
        }

        # Drop asset table
        $session->db->write("drop table NewAsset");

        # Close the session
        $session->var->end;
        $session->close;
        print "Done. Please restart Apache.\n";
}
```

Additional code is added to remove the templates and template folder.

Finally, add a new method which simply updates the templates. This makes it easy to update your templates during development, as well as distribute upgrades to your code.

```
sub upgrade {
   my $config        = $ARGV[0];
   my $home          = $ARGV[1] || "/data/WebGUI";
   my $templateDir = $ARGV[2] || "$home/docs/NewAsset/templates";
   unless ($home && $config) {
        die "usage: perl -MWebGUI::Asset::Wobject::NewAsset -e upgrade
www.example.com.conf\n";
   }

   #Open the session
   print "Updating asset.\n";
   my $session = WebGUI::Session->open($home, $config);
   $session->user({userId=>3});

   #Install the default templates
   importTemplates($session,$templateDir,WebGUI::Asset-
>new($session,"NewAssetFolder00000001"));

   #Commit the version tag containing the templates
   my $tag = WebGUI::VersionTag->new($session, WebGUI::VersionTag-
>getWorking($session)->getId);
   if (defined $tag) {
       print "Committing tag\n";
       $tag->set({comments=>"Template updated by Asset Upgrade Process"});
       $tag->requestCommit;
   }

   #Close the session
   $session->var->end;
   $session->close;
   print "Done.\n";
}
```

The additional methods used to import templates are displayed below. Please note that this is only one way you might accomplish this. WebGUI used to use an import system similar to this one, but now imports packages of templates directly, allowing you to make changes directly on the site and distribute them in WebGUI native package format.

```
#-------------------------------------------------------------------
sub importTemplates {
   my $session     = shift;
   my $templateDir = shift;
   my $folder      = shift;
   my $quiet       = 0;
```

```
#Open the template directory
return 0 unless (opendir (DIR,$templateDir));

#Set the slash variable depending on the operating system
my $slash;
if ($^O =~ /^Win/i || $^O =~ /^MS/i) {
    $slash = "\\";
} else {
    $slash = "/";
}

#Read the files in the template directory
my @files=readdir(DIR);
closedir(DIR);

#For each file in the template directory
foreach my $file (@files) {
    #Skip it if the file isn't valid
    next if ($file eq "." || $file eq ".." || !isValidTemplateFile($file));
    #Create a path to the file and open it
         my $pathToFile = $templateDir.$slash.$file;
         unless ( open (FILE, $pathToFile) ) {
             print "Could not open $pathToFile.  Skipping Template $!\n" unless ($quiet);
             next;
         }
         #Set up some defaults
         my $title        = "Default Template";
         my $namespace    = q{};
         my $assetId      = q{};
             my $settingId    = q{};
         my $templateFile = q{};
             my $headBlock    = q{};
         my $head         = 0;
         #Loop over each line of the template and process meta data, head, and body
         while (<FILE>) {
             my $line = $_;
             my $char = "#";
                 if($line =~ m/^$char[^=<]+=/){
                       my $setting = substr($line,1);
                             my ($key,$value) = split("=",$setting);
           $key = lc($key);

           if($key eq "namespace") {
```

```
                        $namespace = trim($value);
                    }
        elsif($key eq "title") {
                        $title = trim($value);
                    }
        elsif($key eq "assetid") {
                        $assetId = trim($value);
                    }
        elsif($key eq "settingid") {
            $settingId = trim($value);
        }
        next;
                }
    elsif ($line =~ m/^~~~$/) {
                $head = 1;
        next;
                }
    elsif ($head) {
                $headBlock .= $line;
                }
    else {
                $templateFile .= $line;
    }
      }
      #Check for errors in the metadata
      if($assetId eq "") {
          print "No Asset Id specified.  Skipping Template $file\n" unless ($quiet);
                next;
      }

      unless(isValidAssetId($assetId)) {
          print "Skipping $file.  Invalid assetId\n" unless ($quiet);
                next;
      }

      if($namespace eq "") {
          print "No Namespace specified.  Skipping Template $file\n" unless ($quiet);
                next;
      }
      #Process the templates
      print "Processing Asset $assetId\n" unless ($quiet);

      my $tmpl = WebGUI::Asset::Template->new($session,$assetId);
```

```
        if($tmpl){
                  print "Asset $assetId found.  Updating ... \n" unless ($quiet);
                  $tmpl->update({
                           template  => $templateFile,
                headBlock => $headBlock,
                                        namespace => $namespace,
                                        title     => $title,
                                        menuTitle => $title
                        });
                    }
        else {
                  print "Asset $assetId not found.  Creating ... \n" unless ($quiet);
                  $folder->addChild({
                    className   => "WebGUI::Asset::Template",
                namespace   => $namespace,
                title       => $title,
                        menuTitle   => $title,
                        ownerUserId => "3",
                        groupIdView => "7",
                        groupIdEdit => "4",
                        isHidden    => 1,
                template    => $templateFile,
                headBlock   => $headBlock,
            }, $assetId);
        }

        #Set the asset Id in the settings table if set
        if($settingId) {
            my ($exists) = $session->db->quickArray("select count(*) from settings where
name=?",[$settingId]);
            if($exists) {
                $session->setting->set($settingId,$assetId);
            }
            else {
                $session->setting->add($settingId,$assetId);
            }
        }
   }
   return 1;
}

#-------------------------------------------------------------------
sub isValidTemplateFile {
    my $filename = $_[0];
    my $quiet = 0;
    unless ($filename =~ m/(.*)\.(.*?)$/) {
        print "Skipping $filename.  Invalid File Format\n";
              return 0;
    }
    my $extension = $2;

    unless (lc($extension) eq "tmpl") {
        print "Skipping $filename.  Invalid template extension\n" unless ($quiet);
              return 0;
    }
    return 1;
}

#-------------------------------------------------------------------
sub isValidAssetId {
    my $assetId = $_[0];
    unless (length($assetId) <= 22) {
              return 0;
    }
    return 1;
}

#-----------------------------------------
sub trim {
    my $string = $_[0];
    $string =~ s/^\s+//;
    $string =~ s/\s+$//;
    return $string;
}
```

Appendix 2: Writing Patches for WebGUI

As you continue to develop applications or core modules for WebGUI, you may find yourself in the precarious situation of needing to update existing code or apply changes to core prior to an update. A good way to do this is to create a patch file and apply the patch to existing source.

Before getting started with writing patches, you'll need two packages: GNU patch and GNU diffutils. Your operating system vendor should provide these as part of the default installation. If not, you'll typically find pre-compiled packages for them in your operating system's package collection. If neither option is available, you can learn more about the packages at their respective homepages. Learn more about GNU patch at http://www.gnu.org/software/patch/ ; learn more about GNU diffutils at http://www.gnu.org/software/diffutils/ . Windows users can download binary packages at http://gnuwin32.sourceforge.net/ . Before using either command for the first time, it is wise to read the documentation included with each.

diff and patch work together to allow developers to distribute changes to existing code. diff compares two files, or directory trees, given as command line arguments and outputs a summary of how the two differ from one another. This summary is encoded in one of several formats. By default, diff outputs a format compatible with the POSIX standard to the standard output. However, developers frequently use the context format (-c command line option) and the unified format (-u command line option) for patches intended for distribution to other developers.

This is only the first part of the process, however. One needs a program to apply the changes documented in the diff file. This chapter uses the patch program (first developed by Larry Wall, who also created Perl). patch reads the output of the diff command on standard input by default. patch has many options, including the -p option. This specifies how many directories to remove from the file paths in the diff output, used for locating the original and modified files on disk. See patch's documentation for more information.

Developing Against a Release

To develop new code based on the version of WebGUI you have installed, you'll need to make a copy of the existing code. This serves two purposes. First, it serves as a backup copy in case you somehow break WebGUI in a way you cannot fix. Second, it creates a pristine copy against which you'll run the diff command.

To create a patch file for WebGUI, start by making a copy of the WebGUI core you'll be changing.

```
cp -R /data/WebGUI  /data/WebGUIOrig
```

Once you've made all of the changes to source, create a patch or diff file containing all of the changes made to code.

```
cd /data/WebGUI
diff -urN  /data/WebGUIOrig . > mypatch.diff
```

Before applying the patch to the new code, it is wise to verify that the patch will apply correctly. patch provides a command line option (--dry-run in newer versions, --check in older versions) to verify that the patch will apply correctly. For example:

```
patch -p0 --dry-run < /path/to/mypatch.diff
```

The output will be the same as if --dry-run had not been specified. patch will remain mostly quiet upon success, and will complain about errors. However, no files will be written or changed with the --dry-run option present.

This file can then be moved to a matching WebGUI distribution and applied to the source as follows:

```
cd /data/WebGUI
patch -p0 < /path/to/mypatch.diff
```

Patch will attempt to apply all of the changes specified in the diff file created to existing source, making it an extremely convenient way to patch for WebGUI.

Developing With the Latest Code

In addition to modifying code in the installed version of WebGUI, you can also develop against the latest code from WebGUI's Subversion repository. If you want to contribute new features to WebGUI, this is the way to go.

The procedure is largely similar to that described above, with a couple of exceptions. First, there is no need to make a copy of the source code. Your backup and basis of comparison live in the Subversion repository. Second, use svn diff rather than diff:

```
cd /data/WebGUI
svn diff > mypatch.diff
```

Patching remains the same. Remember to frequently update your checkout so that you're not developing against old code.

Index

Index