BigWorld Web Integration Reference

BigWorld Technology 1.9.1. Released 2008.

Software designed and built in Australia by BigWorld.

Level 3, 431 Glebe Point Road
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Overview

This document describes extensions to the popular scripting languages PHP and Python that allow for BigWorld functionality to be integrated into a web server. This document assumes familiarity with either PHP or Python, and elaborates on a Linux-based Apache PHP/Python web server setup (related to the *LAMP architecture*). It also assumes familiarity with concepts presented in the document Server Programming Guide.

For an example on how to implement a web interface to a running BigWorld game cluster and a web auctioning system, see the document How To Implement Web Integration.

Using these extension modules, a web interface can be constructed that accesses script-level BigWorld functionality. This can then be delivered to devices that are web-aware, such as mobile phones, PC browsers, PDAs, and so on.

It should also be noted that while this interface was designed around the needs of a web scripting interface, it is not web specific in that it may be imported by any Python 2.5 (or PHP) instance and used whenever you require an external scripting interface to the base entities on your server. Examples include custom administrative tools and statistic gathering scripts.

1.1. Security

Web security should be a part of all web applications. Therefore, when implementing a BigWorld-aware web application, care must be taken to ensure that users are not able to access privileged information or have unlimited privileged access to the game script interface.

From a low-level security point of view, Apache supports HTTPS transport that is transparent to modules used for PHP and mod_python scripting — for details on how to enable this feature, see the Apache documentation .

From a scripting point of view, much of what is relevant to other web applications with regards to security applies equally to BigWorld-aware web applications. Because the web integration module must be run inside the cluster, care must be taken when designing interfaces to the game. For example, the standard for web applications is to not expose the database backend to users by giving them access to executing raw shell commands or SQL statements. In the same way, do not give users inappropriate privileged access to the BigWorld backend by giving them the ability to run arbitrary script commands. The web integration module does not have the same concepts of Areas of Interest or client controlled entities, so extra care must be taken when accessing game state using this interface.

Chapter 2. Remote Methods with Return Values

Within BigWorld, remote Python method calls are one-way and asynchronous. That is, methods immediately return control back to the calling process. For return values, a callback method is usually defined that is also one-way.

In a web server context, the web server process is only active during the processing of a web request, so that it necessarily has to block if it is to receive return values back from the BigWorld server cluster.

The mechanism of blocking methods with return values is useful here because calls to remote methods block the web server process until a response is received, or the remote method experiences a time-out condition. Return value methods are currently only implemented on base methods, and have the following structure in the entity definitions file:

```
<root>
    ...
    <BaseMethods>
    ...
        <myReturnValueMethod>
            <Arg> INT32  </Arg>  <!-- first argument -->
            <Arg> STRING </Arg>  <!-- second argument -->
            <ReturnValues>
                <someList>   ARRAY <of> INT32</of> </someList>
                <someString> STRING               </someString>
            </ReturnValues>
        <myReturnValueMethod>
    ...
    </BaseMethods>
    ...
</root>
```

Excerpt from an entity definitions file showing a return-value method

In the above example, the return value method is called `myReturnValueMethod`, and it has two arguments — the first is typed as a `INT32`, and the second as a `STRING` type. It returns two values — the first is named `someList` and is typed as an `ARRAY` of `INT32`s, and the second is named `someString` and is typed as a STRING.

Executing these remote methods from the web server context would be carried out in the following manner in mod_python:

```
playerBob = BigWorld.lookUpEntityByName( 'Avatar', 'bob' )
result = playerBob.myReturnValueMethod( 35, 'example' )
req.write( "someString = %s" % result['someString'] )
for listElement in result['someList']:
    req.write( "listElement: %d", listElement )
```

Executing remote methods in mod_python

The actual server-side implementation of remote methods with return values are implemented as base entity methods on the BaseApp (*e.g.*, `base/Avatar.py`). These implementations differ from regular remote methods without return values in that an extra response object is created for each request and passed to the function as an extra parameter.

This response object has attributes corresponding to each return value in the method definition, which the implementation of the method must set values for. When the scripting code has prepared the response back,

it must invoke the `done()` method on the response object for the response to be sent back to the calling context of the web server.

Method response objects cannot be passed as parameters to other server components (*i.e.,* other BaseApps or CellApps), but can be passed around within the same BaseApp.

```
import BigWorld

class EntityType:
    ...
    def myReturnValueMethod( self, response1, argInt32, argString ):
        # The arguments argInt32 and argString match those arguments
        # defined in the .def file.
        ...

        # Set some values of the response, the attributes match
        # those that are defined in the .def file.
        response.someList = [1, 3, 5, 7, 11]
        response.someString = "This is my string"

        # Send the response back, we are done with it, and the web
        # integration module is blocking, pending this response.
        response.done()2
```

Server-side implementation of a remote method

1 The `response` object being passed.
2 The response object `done()` method being called.

Chapter 3. Logging on Entities

When a player logs on with the web module, and the player's entity was not previously loaded from the database, then the entity is loaded onto the least loaded BaseApp. A reference to the newly created or already existing entity base can then be looked up using one of the lookup methods present in the API.

If the entity is already logged on, for example, via the client, then the player entity has already been loaded and will have the `BigWorld.onLogOnAttempt()` method called on the entity base (for details, see the BaseApp Python API documentation). It is called with the parameters `ip`, `port` and `password`. If the logon attempt is from a web integration gateway, then the IP and port will be equal to 0.

In script, the entity must deal with multiple logons from the client through the LoginApp versus multiple logons from the web integration gateway.

There is a facility for ensuring an entity is not destroyed prematurely when accessing entity methods on a mailbox, by sending keep-alive messages and specifying an interval for script callbacks when no keep-alive messages have been sent in that interval (for details, see *Keep-alive Messages* on page 7).

Chapter 4. Keep-alive Messages

Mailboxes to entities residing on a BaseApp should exist for as long as they are needed. However, these same entities may be player entities controlled by the BigWorld client. These two different usages need to be reconciled when it comes to managing entity lifetimes — for example, if the client disconnects while the game's web interface is still using the player's base entity, this entity should not be destructed until the game's web interface has finished with it.

Another example is if the player is currently not logged in via the BigWorld client. If the player does not explicitly log out, we would want to clean up that mailbox reference after an inactivity period.

A solution to this problem is to use a keep-alive interval for mailboxes that require it, so that the entity stays around even if the client has disconnected (destruction of the base entity is the normal course of action). If a keep-alive interval is set on a mailbox, then each time a method is called on that mailbox, it causes the keep-alive interval for that entity to be refreshed.

When a base entity enters a keep-alive interval, the base entity script method `onKeepAliveStart()` is called. When the keep-alive interval expires without any refreshing from the web integration gateway, it is assumed to be no longer required, and so `onKeepAliveStop()` is called in Python script. The base entity script should decide whether it is still required, and should destroy itself.

Keep-alive intervals can be used with HTTP session timeouts so that players have to re-login after a certain inactivity period to create a new HTTP session. Keep-alive intervals should be set to be equal to session durations or longer.

Chapter 5. Threading Issues
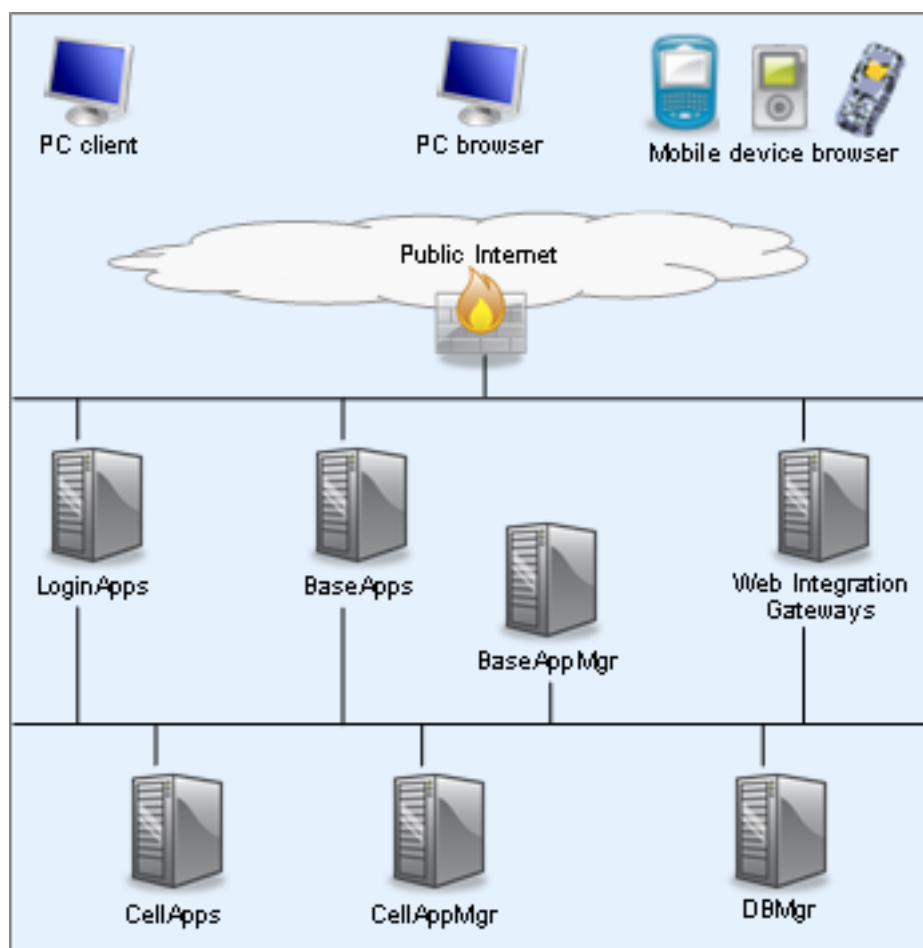
The BigWorld Web Integration module is required by PHP to not use any threading. When used with Apache's default pre-forking multi-processing module, each child fork of Apache contains one instance of the BigWorld module loaded. PHP currently does not support the use of threaded Apache multi-processing modules, and so the BigWorld Web Integration module also does not support threading.

Chapter 6. Deployment

This section deals with the recommended setup for adding web integration to a running BigWorld cluster.

Machines that are dedicated hosts for the web integration modules (we will call them *Web Integration Gateways*), should reside in the same subnet as the rest of the server cluster, and should have public IP addresses.

Multiple web integration gateways can work side by side, with implementation of load balancing done as for other web services, e.g. DNS round-robin.

Example deployment configuration

Chapter 7. Python

7.1. Apache/mod_python

An extension called mod_python is available for Apache HTTP servers that allows for web scripting using Python. There are many frameworks that can use Apache/mod_python as the server base architecture (*e.g.,* CherryPy-based frameworks such as TurboGears), although not using Apache/mod_python is also an option (*e.g.,* using the CherryPy's Python-based web server).

7.2. Building the Module From Source

The sources for the Python extension module are located in `bigworld/src/server/web/python`.

To build the module from source, follow the steps below:

▪ Change to the source directory.

▪ Run `make` to build the module — this should result in the dynamic library file `BigWorld.so` being created in `bigworld/bin/web/Hybrid`.

By default, the source includes header files from the version of Python shipped with BigWorld (Python 2.5). When running with older Python versions (*e.g.,* Python 2.4), warning messages such as the following may appear:

```
__main__:1: RuntimeWarning: Python C API version mismatch for module BigWorld:
 This Python has API version 1012, module BigWorld has version 1013.
__main__:1: RuntimeWarning: Python C API version mismatch for module Math:
 This Python has API version 1012, module Math has version 1013.
__main__:1: RuntimeWarning: Python C API version mismatch for module ResMgr:
 This Python has API version 1012, module ResMgr has version 1013.
__main__:1: RuntimeWarning: Python C API version mismatch for module _BWp:
 This Python has API version 1012, module _BWp has version 1013.
```

Example warning messages that might be displayed when running older Python versions

7.3. Testing the Module

To test the module, change to the output directory `bigworld/bin/web/python/Hybrid`, then run the following command:

```
python2.5 -c "import BigWorld"
```

If no error messages are issued, then it means that you have successfully built the Python module.

7.4. BigWorld UID and Resource Paths

BigWorld processes from a particular user ID can only communicate with other BigWorld processes from that same user ID, and this applies to the Python web integration module. You can override this by changing the `UID` environment variable before importing `BigWorld`, as illustrated below:

```
import os
os.environ['UID'] = 500 # substitute with desired user ID
```

```
import BigWorld
```

Changing `UID` before importing the `BigWorld` module

The BigWorld resource paths also need to be accessible to the web integration module, and they have to be matched to the resources that the server is using. The resource paths are read from the `$(HOME)/.bwmachined.conf` configuration file (for details on BWMachined configuration, see the document Server Overview's section Server Components ➔ BWMachined).

As with with the `UID` environment variable, you can override the `HOME` environment variable to determine where to load `.bwmachined.conf` from.

7.5. Python API

The functions and attributes available in the BigWorld Python API are listed below:

▪ `BigWorld.logOn( username, password, allow_already_logged_on=False )`

Logs on a username/password combination, and either returns `None` (on success) or raises an exception (on failure). The exception type reflects the kind of error that occurred, and can be one of the following values: `IOError`, `ValueError` and `SystemError`.

If the `allow_already_logged_on` parameter is `True`, then this function does not raise an exception if the BigWorld server cluster reports that the user is already logged on.

▪ `BigWorld.lookUpEntityByName( entityType, entityName )`

Queries the DBMgr to look up an entity base for a given entity type and entity name — if the entity exists and the entity base is loaded, then it returns a mailbox object pointing to the entity base that can be used for invoking remote methods. If the entity exists, but is not loaded, then the function returns `True`; otherwise it returns `False`.

▪ `BigWorld.setDefaultKeepAliveSeconds ( defaultKeepAliveSeconds )`

Sets the default keep-alive interval for new mailboxes.

This interval can also be adjusted per-mailbox through the `keepAliveSeconds` attribute.

▪ `BigWorld.setNubPort( port )`

Recreates the Mercury communications nub object and resets any addresses to BigWorld services that it has cached.

The nub object has its port set to the given port, unless the given port is zero, in which case a random port is chosen.

▪ `MailBox.keepAliveSeconds`

The keep-alive interval (in seconds) of the specified entity mailbox.

```
mailbox.keepAliveSeconds = newKeepAliveSeconds
```

▪ `MailBox.serialise()`

Serialise this mailbox to a string.

▪ `BigWorld.deserialise( serialisedMailbox )`

Deserialise the serialised mailbox string to a mailbox.

7.6. Remote Methods

Invocations of remote methods with defined return values return a dictionary of those return values with the names of the defined return values as keys.

```
<root>
    ...
    <BaseMethods>
        ...
        <getGoldAndInventory>
            <Arg>  INT32  </Arg>  <!-- bag ID -->
            <ReturnValues>
                <goldPieces>    UINT32                            </goldPieces>
                <inventoryList> ARRAY <of> INVENTORY_ITEM </of> </inventoryList>
            </ReturnValues>
        </getGoldAndInventory>
        ...
    </BaseMethods>
    ...
</root>
```

Example remote entity method definition

Remote methods can be invoked on a mailbox directly, as illustrated below:

```
player = BigWorld.lookUpEntityByName( 'Avatar', 'bob' )  # mailbox to Avatar
 'bob'
bagID = 0
result = player.getGoldAndInventory( bagID )      # argument is integer
gold = result['goldPieces']                       # integer
inventoryList = result['inventoryList']           # list of inventory items
```

Example invocation of a remote entity method with return values in Python

The remote method `getGoldAndInventory` is called using the previous looked-up mailbox of the Avatar entity named `bob`. The lookup involves a query to DBMgr, which gives the mailbox (which locates the entity object amongst the BaseApps) and the method call itself is directed to the entity base, through the BaseApp that it resides on.

7.7. Mailboxes and Persistent Session Storage

Apache/mod_python provides a mechanism for storing Python objects in the server-side persistent session variables. This makes it unnecessary to look entities up on every page request, when the mailbox can be stored in the session variables on look-up, and restored on page load. This also means that entity mailboxes are shared across all child processes of Apache. You use the mailbox's `serialise()` method to get the mailbox serialised form as a string, and save this string to the persistent session. On subsequent requests, you can re-constitute the mailbox via the `BigWorld.deserialise()` method.

An example is displayed below:

```
from mod_python import Session
...
import BigWorld
...
```

```
def handler( req ):
    session = Session.Session( req )

    if not 'player_mailbox' in session:
        # they aren't logged in yet, login if they have supplied credentials
        if not 'username' in req.fields or not 'password' in req.fields:
            req.write( "login credentials required" )
            return

        playerName = req.fields['username']
        password = req.fields['password']

        try:
            BigWorld.logOn( playerName, password )
        except:
            req.write( "Login failed" )
            return

        mailbox = BigWorld.lookUpEntityByName( 'Avatar', playerName )
        if type( mailbox ) == bool:
            req.write( "Login error" )
            return

        session['player_mailbox'] = mailbox.serialise()[1]
    else:
        # we are already logged in, mailbox is in session
        mailbox = BigWorld.deserialise( session['player_mailbox'] )[2]

    res = mailbox.getGoldAndInventory()
    ...
```

[1] The `serialise()` method is called here on a mailbox that has just been retrieved via `BigWorld.lookUpEntityByName()`. The entity was created when authentication of `BigWorld.logOn()` succeeded. The string is stored away in the persistent session under the key `player_mailbox`.

[2] The `deserialise()` method is called here on the serialised string stored in the persistent session under the key `player_mailbox`, and the mailbox is re-constituted.

7.8. Example Scripts

Example scripts demonstrating how to use the BigWorld Python extension module to integrate with mod_python are located in `fantasydemo/src/web/mod_python`.

To try out these scripts, you can add the following lines to your Apache configuration, changing the root of the the paths (fill in the ellipsis below):

```
# Create a convenient path to these scripts
Alias /bw-python-example ...fantasydemo/src/web/mod_python
# Directory containing game scripts
<Directory ...fantasydemo/src/web/mod_python >
    # Automatically look up index.py
    DirectoryIndex index.py
    # Allow MultiViews to make "page" reach "page.py"
    Options Indexes MultiViews
    # Filter all python scripts through mod_python
    AddHandler mod_python .py
    # Use module bwmain to execute our scripts
    PythonHandler bwmain
    # Allow debugging info
```
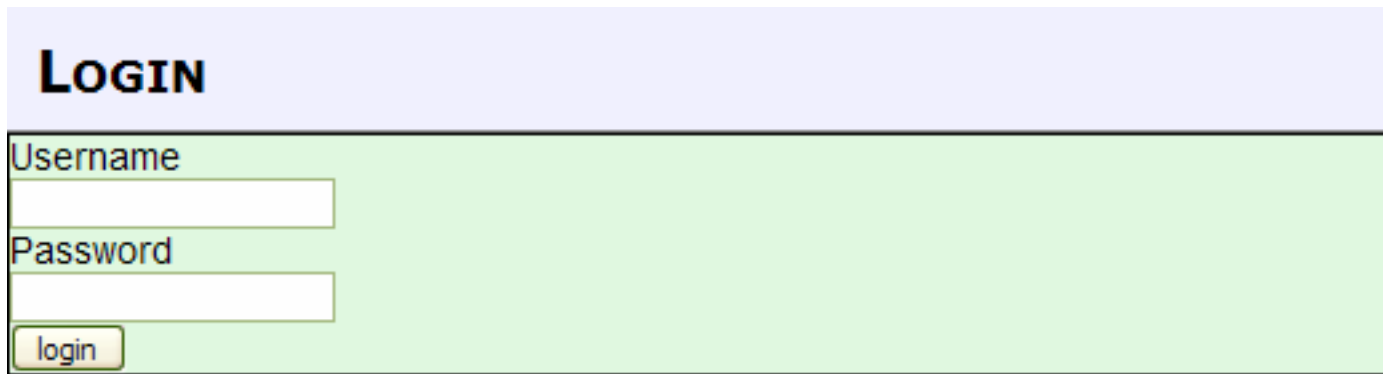
```
      PythonDebug On
      # Use Python sub-interpreter called "bigworld"
      PythonInterpreter bigworld
      # Let Python find its modules
      PythonPath "sys.path + ['...fantasydemo/src/web/mod_python',
  '...bigworld/bin/web/Hybrid']"
      # These Python scripts output HTML
      AddType text/html .py
</Directory>
```

Please consult the Apache manual and/or your distribution's documentation on how to add configuration directives to your Apache configuration.

You also must edit `fantasydemo/src/web/mod_python/lib/BigWorldConstants.py` to set the correct UID to connect to your server (see "BigWorld UID and Resource Paths" on page 10).

The example script above assumes that the mod_python module is installed and loaded. For more details, see the Apache configuration file documentation and the mod_python installation documentation. The configuration template above will make the example scripts available at `http://<hostname>/bw-python-example`.

mod_python example scripts login page

Chapter 8. PHP

A PHP module is provided to interface with a BigWorld server through script. PHP is a very popular open-source scripting language for web development. This module is designed to be used under Linux with Apache/mod_php, and in addition, it has also been known to work with the PHP Linux command-line interpreter as well.

8.1. Dependence on Python Module

The PHP module is actually a wrapper around the Python module described above, and thus it requires the Python module to be accessible for use. For details on how to install it (and, if necessary, build it), see "Building the Module From Source" on page 10 .

8.2. Independence From Apache

This installation guide supposes that you will be using Apache, but this is not a hard limit — you can use the BigWorld PHP Web Integration Module with any web server that supports PHP under Linux (*e.g.*,, you could use the PHP CLI interface — this implies that any web server that supports CGI will work).

8.3. Building the Module From Source

8.3.1. Requirements

Building the PHP module requires the PHP development environment — either the PHP source distribution tarball or the PHP source distribution packaged for your distribution (*e.g.*, `php-devel` in Fedora Core or `php5-dev` in Debian). For details on how to install these packages, see your distribution's documentation.

8.3.2. Building a PHP-compatible Python Distribution

Due to requirements of the PHP runtime, the PHP module is required to link against a version of Python 2.5 that is built without thread support (see *Threading Issues* on page 8). Also, the BigWorld Python module, which is imported by the BigWorld PHP module, requires that Unicode UCS4 support is enabled. Note that these requirements can be different to whatever stock Python package is usually distributed with supported Linux distributions, so a separate local build of Python 2.5 is necessary in order to build the PHP shared library.

Please note that you cannot use the Python 2.5 sources from `src/lib/python`, as it has been modified for use with the other BigWorld server components and client.

You can download the Python 2.5 source distribution from `http://www.python.org`. Once it has been downloaded, decompress the Python 2.5 tarball, then build the Python 2.5 distribution, as per below:

```
$ ./configure EXTRA_CFLAGS=-fPIC CFLAGSFORSHARED=-fPIC --enable-unicode=ucs4
  --without-threads --prefix=/usr/local/python2.5
...
$ make
...
$ sudo make install
...
```

The above example also specifies the `--prefix` option where the locally built Python 2.5 installation to reside (`/usr/local/python2.5`). This path needs to match where the PHP build process expects the Python 2.5 distribution to be, so ensure that the path given by `--prefix` matches what is in the `config.m4` file (for details, see "Building the PHP Module" on page 16).

8.3.3. Building the PHP Module

You need to tell the PHP build process where to find the Python source tree by editing `bigworld/src/server/web/php/config.m4`, as displayed in the excerpt below:

```
   PHP_REQUIRE_CXX()
# Put the path to your own python2.5 include path here
   PHP_ADD_INCLUDE(/usr/local/python2.5/include/python2.5)
   PHP_SUBST(BIGWORLD_PHP_SHARED_LIBADD)

# Put the path to your own python2.5 static library path here
   PHP_ADD_LIBRARY_WITH_PATH(python2.5,
 /usr/local/python2.5/lib/python2.5/config, BIGWORLD_PHP_SHARED_LIBADD)
```

Excerpt from `bigworld/src/server/web/php/config.m4` — Defining Python's source tree

You also need to run a command called `phpize` in the module source's root directory at `bigworld/src/server/web/php` — an example output is displayed below:

```
$ phpize
Configuring for:
PHP Api Version:         20020918
Zend Module Api No:      20020429
Zend Extension Api No:   20021010
```

Example output from `phpize`

This will generate some files for facilitating the build process, including a `configure` script. You may then build the module and install it as follows:

```
$ ./configure
...
$ make install
...
```

Configuring, building and installing the PHP module

The above commands compile the module and install it as a PHP extension in the directory where PHP expects it. Note that the last command (`make install`) usually requires superuser privileges.

8.4. Installation

8.4.1. Requirements

You will need both the Python extension module and the appropriately versioned PHP extension module for your distribution of PHP. For details on installation of PHP, see the PHP Installation documentation.

8.4.2. Module Loading

The BigWorld module performs some initialisation of the network interfaces and structures used to talk to a BigWorld server cluster, and it contains some static data that persists over the lifetime of that module — loading it at start-time ensures that this step has been done once and only once for the lifetime of the PHP process. The alternative of not loading the module at start-time implies that the module needs to be loaded every time before use, thus duplicating this load-time initialisation step unnecessarily.

> **Note**
>
> If PHP is invoked as a CGI process rather than as part of the web server (*e.g.*, PHP as an Apache Dynamic Shared Object (DSO) residing within the Apache web server process), then this implies that the `bigworld_php.so` PHP extension module is loaded each time that PHP is invoked.
>
> The recommended setup is for PHP to be a Dynamic Shared Object and loaded into the Apache process to prevent this from happening. For details, see the PHP and Apache HTTP Server documentation.
>
> Both Fedora Core and Debian distributions provide binary packages for PHP as a DSO. For details on installation, see your distribution's documentation.

8.4.3. Configuration

Packaged versions of PHP have a configuration file directory (*e.g.*, `/etc/php.d` on Fedora Core distributions or `/etc/php5/conf.d` on Debian), for which each file contained is included as part of the PHP configuration. Source distributions of PHP can specify this when running the configure script before building.

A sample configuration file is included in `bigworld/src/server/web/php/sample.ini` and is included below:

```
;
; Sample PHP configuration for the bigworld module
;
; Put this file in your PHP configuration files directory, e.g. /etc/php.d or
 /etc/php/conf.d
; and modify it with your correct settings.
;

; PHP directive to load the extension module. This requires that the PHP
 module be
; built and installed into wherever PHP is expecting PHP extension modules to
 reside
extension=bigworld_php.so

[bigworld]
; Change this to be the UID that your server is running under.
bigworld.uid = 500

; Change this to be the location of BigWorld.so. Typically
 bigworld/bin/web/Hybrid.
; NOTE: This has to be set correctly for the PHP module to function!

bigworld.additional_python_paths = /../bigworld/bin/web/Hybrid

; Increase for more debugging.
bigworld.debug_level = 0
```

Sample configuration file `bigworld/src/server/web/php/sample.ini`

Some distributions have PHP packaged so that configuration can be placed in `/etc/php/conf.d` or similar directory. It is usually good practice to place module-specific configuration in its own file in such a directory (*e.g.*, `bigworld.ini`).

8.4.4. Configuration Directives

The configuration directives used in the `php.ini` file that are required for using the PHP module are described in the list below.

- **bigworld.uid**

  This directive specifies the user ID of the BigWorld server to communicate with. This also indirectly determines the resource path, as it is read from `~/.bwmachined.conf`. This is required because Apache is usually run as a special web account user (e.g. www-data).

- **bigworld.additional_python_paths**

  This directive specifies the additional python paths to include in the Python interpreter's `sys.path`. In practice, this is used to specify the location of the BigWorld Python extenstion module in `$(MF_ROOT)/bigworld/bin/web/Hybrid`. This path to `BigWorld.so` must be included in `bigworld.additional_python_paths`.

- **bigworld.debug_level**

  This optional directive specifies the level at which the error log entries are generated from use of the PHP module to the PHP error log. The possible values for this are:

  - **0** — Debug level: `ERROR`

    Only errors are output to the log.

  - **1** — Debug level: `INFO`

    Only errors and information messages are output to the log (information messages include remote method calls and their parameters).

  - **2** — Debug level: `TRACE`

    Currently, this is the same as the `INFO` level.

  - **3** — Debug level: `MEMORY`

    Additional information about memory allocations and deallocations are output to the log, as well as messages for lower log levels.

8.5. Testing

To test the configuration changes above, you can use the `phpinfo()` function, which prints PHP-related information, including any modules that have been loaded and their configuration directive values.

The easiest way to do this is to create a PHP script as illustrated below and use a browser to view it:

```php
<?php
  phpinfo();
?>
```

Testing PHP configuration changes

PHP Version 5.2.0-8+etch7

System	Linux raxa 2.6.18-4-k7 #1 SMP Wed May 9 23:42:01 UTC 2007 i686
Build Date	Jul 2 2007 21:30:29
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/apache2/php.ini

Example `phpinfo` output

The BigWorld PHP module should be listed amongst the other PHP loaded modules, as illustrated below.

bigworld_php

Directive	Local Value	Master Value
bigworld.additional_python_paths	/home/dominicw/bw-19d/bigworld/bin/web	/home/dominicw/bw-19d/bigworld/bin/web
bigworld.debug_level	1	1
bigworld.uid	605	605

Example `phpinfo()` output — BigWorld module configuration

8.6. Mailbox Session Storage

As they stand, Python objects represented as PHP resources do not persist across page requests. PHP resource objects are typically discarded at the end of the page request. Without persistence, mailboxes would have to be looked up each time a page was loaded that required remote methods to be called.

In mod_python, when Python objects (including mailboxes) are set in the session variables in the form of serialised strings (or *pickles*, in Python parlance), they are serialised when the session variable is set and then deserialised at page load into mailboxes. PHP has a similar session mechanism accessed via the `$_SESSION` superglobal variable, after first initialising the session by calling PHP's `session_start()`.

There are two functions in the API to facilitate a similar mechanism where mailboxes are serialised and deserialised: `bw_serialise()` and `bw_deserialise()`. The string returned from `bw_serialise()` applied on a mailbox can be stored in a PHP session, and the mailbox can then be restored by applying `bw_deserialise()` — it returns a PHP resource which can be used in `bw_exec()` to call remote methods.

```
session_start(); // enable PHP sessions
...
$mailbox = bw_look_up_entity_by_name( "Account", $username );
$_SESSION['mailbox'] = bw_serialise( $mailbox ); // serialises the mailbox
 into a string using Python pickling
```

Example of saving a mailbox to session storage in PHP

```
$mailbox = bw_deserialise( $_SESSION['mailbox'] ); // deserialises the mailbox
 pickle string
$res = bw_exec( $mailbox, ... ); // call some method on the deserialised
 mailbox
```

Example of retrieving a mailbox from session storage in PHP

8.7. Remote Methods

Invocations of remote methods with defined return values return an associative array of those return values with the names of the defined return values as keys.

In the examples below, the remote method `getGoldAndInventory()` is called using the previous looked-up mailbox of the Avatar entity named `bob`. The lookup involves a query to the DBMgr, which returns the mailbox (which locates the entity object amongst the BaseApps). The method call takes an argument of `INT32`, and is directed to the Avatar entity base specified by the looked-up mailbox. The method invocation returns back a PHP array that has keys `goldPieces` and `inventoryList` set to the return values from the BigWorld server.

```
<root>
    ...
    <BaseMethods>
        ...
        <getGoldAndInventory>
            <Arg>  INT32  </Arg> <!-- bag ID -->
            <ReturnValues>
                <goldPieces>    UINT32                          </goldPieces>
                <inventoryList> ARRAY <of> INVENTORY_ITEM </of> </inventoryList>
            </ReturnValues>
        </getGoldAndInventory>
        ...
    </BaseMethods>
    ...
</root>
```

Example remote entity method definition

```
$player = bw_look_up_entity_by_name( "Avatar", "bob" );  # mailbox to Avatar
 'bob'
$result = bw_exec( $player, "getGoldAndInventory", $player, $bagID );    #
 argument is integer
$gold = $result['goldPieces'];                      # integer
$inventoryList = $result['inventoryList'];          # list of inventory items
```

Example invocation of a remote entity method with return values in PHP

8.8. The PHP BigWorld API

The methods available in the PHP BigWorld API are described in the list below:

▪ bw_exec(*$mailbox*, *$baseMethodName*, ...)

Executes a remote method on an entity mailbox given as a PHP resource, as returned by `bw_look_up_entity_by_name()` and `bw_look_up_entity_by_dbid()`.

This method has a variable size argument lists, which are converted to Python objects before being sent to the Python remote method.

The arguments are defined in the entity definition file, and should match the order in which they appear.

If the remote method contains return values, then these are converted to PHP types. In particular, return values are always returned as a PHP array, with the keys being the return value names as defined in the function specification in the entity definition file

If there are no return values, then `NULL` is returned.

- `bw_get_keep_alive_seconds(` *$mailbox*, *$keepAliveSeconds* `)`

Returns the keep-alive period (in seconds) for the specified mailbox.

- `bw_logon(` *$username*, *$password*, *$allow_already_logged_on=FALSE* `)`

Logs on a user given an input password. The method returns `TRUE` if the logon attempt was successful, otherwise it returns an error message string.

- `bw_look_up_entity_by_dbid(` *$entityType*, *$dbId* `)`

Queries the BigWorld server for a specific entity with the given type and database ID. If the entity exists and an entity base has been loaded into the BigWorld system, then it returns a an entity mailbox that can be used by `bw_exec()` to invoke methods on that entity base.

If the entity exists but has not been loaded, then `TRUE` is returned. If the entity does not exist, then `FALSE` is returned.

- `bw_look_up_entity_by_name(` *$entityType*, *$entityName* `)`

Queries the BigWorld server for a specific entity with the given type and name. If the entity exists and an entity base has been loaded into the BigWorld system, then it returns an entity mailbox pointing to that can be used by `bw_exec()` to invoke methods on that entity base.

If the entity exists but has not been loaded, then `TRUE` is returned. If the entity does not exist, then `FALSE` is returned.

- `bw_set_default_keep_alive_seconds(` *$keepAliveSeconds* `)`

Sets the default keep-alive period (in seconds) for all new entity mailboxes.

- `bw_set_keep_alive_seconds(` *$mailbox*, *$keepAliveSeconds* `)`

Sets the keep-alive period (in seconds) for the specified mailbox.

- `bw_set_nub_port(` *$port* `)`

Recreates the Mercury communications nub object and resets any addresses to BigWorld services that it has cached.

The nub object has its port set to the given port, unless the latter is zero, in which case a random port is chosen.

- `bw_serialise(` *$mailbox* `)`

This method serialises a mailbox to a string that can be used to store away a mailbox into whatever session persistence mechanism your web script environment uses. The mailbox can be re-made using `bw_deserialise()`.

- `bw_deserialise(` *$serialisedMailbox* `)`

This method deserialises a serialised mailbox string into a mailbox.

8.9. Example Scripts

Example scripts demonstrating how to use the BigWorld PHP extension module to integrate with PHP are located in `fantasydemo/src/web/php`.

8.9.1. Requirements

For details on how to set up the requirements of the PHP example scripts, see your Linux distribution instructions. The requirements are listed below:

▪ WURFL — included with the example scripts (for details, see "WURFL" on page 22)

▪ PHP CLI and Apache2 module — the CLI is used to install WURFL

▪ PHP `gd` module — for image operations

▪ Apache (2.2 series)

▪ Running BigWorld server

8.9.2. WURFL

WURFL is a library for reading Wireless Universal Resource Files that describe capabilities of mobile devices and provides support for searching these capabilities based on the HTTP User Agent string. This package needs to be set up for the example scripts — you will need a command-line version of PHP installed. For details on WURFL, see http://wurfl.sourceforge.net.

8.9.3. Installation

Make sure that the BigWorld PHP module has been built (see "Building the Module From Source" on page 15) and is being loaded from startup.

The WURFL cache needs to be updated before the example scripts can be used — to update it, follow the steps below:
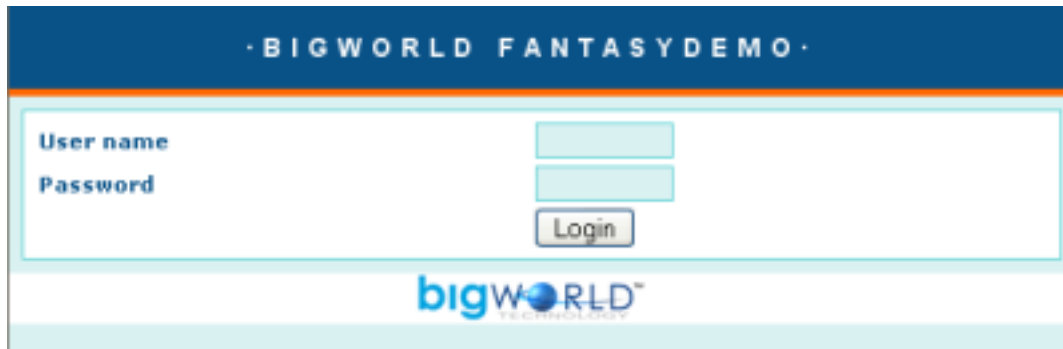
▪ Move to the WURFL directory.

```
cd fantasydemo/src/web/wurfl
```

▪ Run the command below:

```
php update_cache.php
```

▪ Make a symbolic link to `fantasydemo/src/web/php` somewhere in Apache's `DocumentRoot` (*e.g.*, `/var/www/html/fantasydemo`).

▪ Make sure that the Apache configuration directive `FollowSymlinks` is on for the directory containing the symlink, and that the web server user has full read access to the directories and files underneath `fantasydemo/src/web/php` (for more details, see the Apache manual).

▪ Point the browser to the corresponding URL. If you log in using an account already created via the FantasyDemo client, then you should be able to see the character that you created there in a list on successful login. You can then view character statistics and inventory, create auctions from your inventory, check your own auctions, and bid on other player's auctions.

PHP example scripts login page

Chapter 9. Porting to Other Scripting Languages

The PHP module wraps the Python module and maps types between PHP and Python. In theory, this could be done with other scripting languages, for example Ruby.

The source to the PHP and the Python modules is located in `bigworld/src/server/web` under the directories `php` and `python`. You can use the PHP module source as example code to implement an extension module for other scripting languages.

Chapter 10. Caveats

10.1. Using Both the PHP BigWorld Extension and Apache/mod_python Simultaneously

The PHP module works by embedding a Python interpreter into the PHP process, which then forms part of the the Apache HTTPd process tree. This is also the behaviour of the mod_python Apache module, and loading both the PHP BigWorld extension and the mod_python Apache module simultaneously has undefined effects, and is therefore not supported.

10.2. Restarting the Cluster

The web integration module will cache mailboxes between transactions and even sessions for efficiency. However when the server cluster is restarted these mailboxes become invalid. The module must therefore also be reinitialised. The easiest way of doing this is by restarting the web server that loaded it, whenever the cluster is restarted.