Server Web Integration Guide

BigWorld Technology 2.1. Released 2012.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Overview

This document describes how a web interface can be constructed that accesses script-level BigWorld functionality. Web browsers, smart phones and other web-aware devices can then be used to access game functionality. It also demonstrates how to provide a lower level web service interface to the game server that can be used by other services.

This document is broken up into three parts:

1. Exposing the BigWorld server as a web service: this section explains the core elements of BigWorld that are used as the basis for web integration. Web integration is usually implemented using a Service running on ServiceApps. Twisted.Web is used to provide a HTTP interface for making script calls on the server.

2. Using the web service from Apache: this section describes using PHP to implement a website that interfaces with the BigWorld server.

3. Web integration example: this section details the implementation of an Auction House in FantasyDemo. It is intended to be used to demonstrate a website integrated with BigWorld.

Note that while Apache and PHP are used for the implementation of these examples, it is possible to use other languages or servers to implement web integration.

Part I. Exposing the BigWorld Server as a Web Service

Table of Contents

Chapter 2. The TwistedWeb Service

It is often useful to create a Web Service interface to a BigWorld server. This allows other standard services to be used to access game functionality via standard HTTP requests.

One way to expose a web service interface is to use the TwistedWeb Service provided. This uses the Twisted Python framework and its Twisted.Web module to map HTTP requests to script calls on game entities. Any of an entity's methods can be called on it in this way.

See http://twistedmatrix.com/documents/current/web for more detailed information on Twisted.Web.

A BigWorld Service is a scripted object like a Base-only entity. See Server Overview's section *Design Introduction* for more information.

BigWorld provides a standard TwistedWeb Service located at `bigworld/res/scripts/service/TwistedWeb.py`. This service listens for HTTP requests on port 8000. It supports four types of URL paths.

- `db/<dbCommand>?<arguments>` - This is used to invoke a command on the database such as logging on a player's entity.
- `entities_by_id/<entityType>/<databaseID>/<methodName>?<arguments>` - This calls a method on an entity.
- `entities_by_name/<entityType>/<entityName>/<methodName>?<arguments>` - This calls a method on a named entity.
- `global_entities/<globalName>/<methodName>?<arguments>` - This calls a method on an entity in `BigWorld.globalBases`.

The responses to these requests are structured as JSON documents.

2.1. An example

For example, in FantasyDemo, the method `webTestMethod` for the global entity `AuctionHouse` is used to test the functionality of method calls using Twisted.Web.

In `fantasydemo/res/scripts/entity_defs/AuctionHouse.def`, the method is declared as:

```
...
<BaseMethods>
  ...
  <webTestMethod>

    <Args>
      <first_arg> INT32 </first_arg>
      <second_arg> STRING </second_arg>
    </Args>

    <ReturnValues>
      <first_result> INT32 </first_result>
      <second_result> STRING </second_result>
    </ReturnValues>

  </webTestMethod>
  ...
```

And in fantasydemo/res/scripts/base/AuctionHouse.py as:

```
class AuctionHouse( ... ):
    ...
    def webTestMethod( self, first_arg, second_arg ):
        return (2 * first_arg, second_arg.upper())
```

An instance of AuctionHouse has been registered with `BigWorld.globalBases` as `'AuctionHouse'`.

Therefore, requesting:

```
http://machine_name:8000/global_entities/AuctionHouse/webTestMethod?
first_arg=5&second_arg=Test
```

returns the JSON object:

```
{ "first_result": 10, "second_result": "TEST" }
```

2.2. Queries to /db/

There is currently only one command supported by the `/db/` path. This has the form:

```
http://machine_name:8000/db/logOn?username=username&password=password
```

This attempts to log on the user. On success it returns the new entity's type and database id. For example,

```
{ "type": "Account", "id": 2 }
```

This information can then be used to make queries on this entity using the path starting with `/entities_by_id/<entity_type>/<database_id>/<methodName>`.

2.3. Queries to /entities_by_id/

Queries of the form `/entities_by_id/<entity_type>/<database_id>/<methodName>?<args>` can be used to call methods on a specific entity.

For example, requesting:

```
http://machine_name:8000/entities_by_id/Account/2/webGetCharacterList
```

might return:

```
{ "characters": [{"type": "Avatar", "databaseID": 1, "realm": "fantasy",
 "charClass": "ranger", "name": "MyChar"}] }
```

The details of one of the characters on this account can be retrieved using `http://machine_name:8000/entities_by_id/Account/2/webChooseCharacter?name=MyChar&type=Avatar`:

```
{ "type": "Avatar", "id": 1 }
```

2.4. Queries to `/entities_by_name/`

These queries are similar to `entities_by_id` expect that the database string identifier is expected instead of the database id. The `entities_by_id` form is preferred as queries by name need to query the database each time while repeated queries via database id will likely hit a local cache of the entity's mailbox on the ServiceApp. See KeepAlive messages below.

2.5. Queries to `/global_entities/`

These queries allow calling methods on a base entity that has been registered with `BigWorld.globalBases`. The base entity must be registered with a single string as the key. These queries have the form:

```
http://machine_name:8000/global_entities/<global_key>/<methodName>?<args>
```

2.6. Implementation details

The TwistedWeb service is defined in `bigworld/res/scripts/service_defs/TwistedWeb.def`, and its methods are implemented in `bigworld/res/scripts/service/TwistedWeb.py`. Here, the resource tree is built up using Twisted.Web's `putChild` function, which takes as arguments the name of the path segment and the type of the resource that will be returned by a request for it:

```
from TWResources.EntitiesResource import EntitiesByNameResource,
 EntitiesByIDResource
from TWResources.GlobalEntitiesResource import GlobalEntitiesResource
from TWResources.DBResource import DBResource

class TwistedWeb( BigWorld.Service ):
    def __init__( self ):
        root = resource.Resource()
        root.putChild( "entities_by_name", EntitiesByNameResource() )
        root.putChild( "entities_by_id", EntitiesByIDResource() )
        root.putChild( "global_entities", GlobalEntitiesResource() )
        root.putChild( "db", DBResource() )

        reactor.listenTCP( 8000, server.Site( root ) )
        reactor.startRunning()

    def onDestroy( self ):
        reactor.stop()
```

The various resources used by the TwistedWeb service are implemented in the TWResources package, which is located at `bigworld/res/scripts/service/TWResources`.

In order to make use of the TwistedWeb service, it must be given an entry in the *<res>*`/scripts/services.xml` file in your project directory:

```
<root>
    ...
    <TwistedWeb/>
</root>
```

This file contains a list of all Services that will be initialised when a ServiceApp process is started.

See http://twistedmatrix.com for more detailed information on the Twisted Python framework.

2.7. TwistedWeb Error handling

Two-way calls to the game server using the TwistedWeb service will always return a JSON object. If an error occurs, the object that is returned will have a specific error format. It will consist of two fields: a string named `excType` containing the error type, and an array of strings named `args` containing the arguments. The returned document also uses the HTTP error code 403 (Forbidden).

For example, the sample `/db/` queries given in section “Queries to `/db/`” on page 12 could fail in a number of ways. If the account does not exist on the server, the call will return:

```
{ "excType": "BWAuthenticateError", "args": [ "No such user" ] }
```

If the password is invalid, it will return:

```
{ "excType": "BWAuthenticateError", "args": [ "Invalid password" ] }
```

The regular format of error objects returned from TwistedWeb means that differentiating them from successful return objects only requires the caller to check for the existence of the `excType` key.

For details about the different types of errors that can be returned by a two-way call, refer to theServer Programming Guide's section“BWStandardError”.

Chapter 3. Remote Methods, Arguments and Return Values

The `TwistedWeb` service allows calls to one-way and two-way methods on game entities. These can be made using the request paths as described above.

This chapter gives more details about format of the arguments in the query string, the returned results and the supported types.

3.1. Method calls

Any method of a base entity can be called whether it is exposed to clients using the `<Exposed/>` tag or not. The method name is the last part of the URL before the parameter list (i.e. before any `?` character). Care must be taken to ensure that general access to call methods on the `TwistedWeb` service is not given.

To call a method on a cell entity, first call a method on a base entity that returns the result of a call on the cell entity.

3.2. Arguments

The arguments are passed as URL parameters. All arguments are named and so must be named in the entity's `.def` file.

Not all types are supported. Supported types include:

- All integer types - INT8, INT16, INT32, INT64, UINT8, UINT16, UINT32 and UINT64
- All float types - FLOAT32 and FLOAT64
- All string types - STRING, UNICODE_STRING and BLOB
- VECTOR2, VECTOR3 and VECTOR4
- Sequence types - ARRAY and TUPLE of these types except for other sequence types

UNICODE_STRING parameters should be percent-encoded for UTF-8, such as:

```
myString=Japanese%20translation%3A%20%E6%96%87%E5%AD%97%E5%8C%96%E3%81%91
```

This string will result in the argument:

```
Japanese translation: 文字化け
```

BLOB parameters should be passed as a hexidecimal representation, for example:

```
myBLOB=aca3a6
```

VECTOR2, VECTOR3 and VECTOR4 should be passed as a comma separated sequence of floats, for example:

```
myVector=0.1,3.5,-6
```

Sequences can be passed in two ways - either as repeated arguments:

```
myArray=4&myArray=53
```

or as indexed arguments:

```
myArray[0]=4&myArray[1]=53
```

3.3. Return Values

Return values are returned as a JSON document of name/value pairs. These are encoded using the Python `json` module. Any type that can be converted with this module can be used. Additionally `Vector2`, `Vector3`, `Vector4` and `PyArrayDataInstance` data types are converted to Python lists before this conversion.

3.4. One way calls

It is also possible to call one-way methods. On success, the following object is always returned.

```
{"message": "One way call made"}
```

3.5. Errors

Errors are returned as a specially formatted JSON object with an "excType" attribute. Refer to “TwistedWeb Error handling” on page 14 for more information.

Chapter 4. Keep-alive Messages

Mailboxes to entities residing on a BaseApp should exist for as long as they are needed by the web server. However, these same entities may be player entities controlled by the BigWorld client. These two different usages of the same entity must 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. Also, if the player is currently not logged in via the BigWorld client. If the player does not explicitly log out of the web server, we would want to clean up that mailbox reference after an inactivity period.

The solution to this problem is for the `TwistedWeb` service to periodically inform the base entity that it is still interested in it. The entity can then stay around even if the client has disconnected (destruction of the base entity is the normal course of action).

The `TwistedWeb` service keeps a cache of mailboxes. Only mailboxes to entities that have a `webKeepAlivePing()` method are cached. Each time the service is queried that uses one of these mailboxes it is either added to this cache or marked with the time it was last queried.

The service will periodically check all mailboxes in this cache. If the mailbox has not been used for a long time, it is removed from the cache. If it is still active, the `webKeepAlivePing()` method is called on it.

It is okay if the mailbox is removed from the cache too early. In this case, the database will be queried to retrieve the mailbox. This is important when running multiple `TwistedWeb` service fragments on different ServiceApps. Besides a very minor performance impact on the database, it does not matter which ServiceApp is queried. Running the service on multiple ServiceApps is a way to achieve fault tolerance and also scaling beyond a single machine.

This functionality is implemented in `bigworld/res/scripts/service/TWResources/KeepAliveMailboxes.py`. The frequency of the pings is specified by the `CHECK_PERIOD` constant. The amount of time before a mailbox times out is specified by the `TIMEOUT_PERIOD` constant.

For game script, entities can make use of the `KeepAlive` class located in `fantasydemo/res/scripts/base/KeepAlive.py` to implement this functionality. It also has `CHECK_PERIOD` and `TIMEOUT_PERIOD` constants specifying how frequently to check for these entities timing out and how long before they time out. This timeout period should be slightly more than that of `KeepAliveMailboxes.py`.

When accessing this service from a website, 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 or longer than session durations.

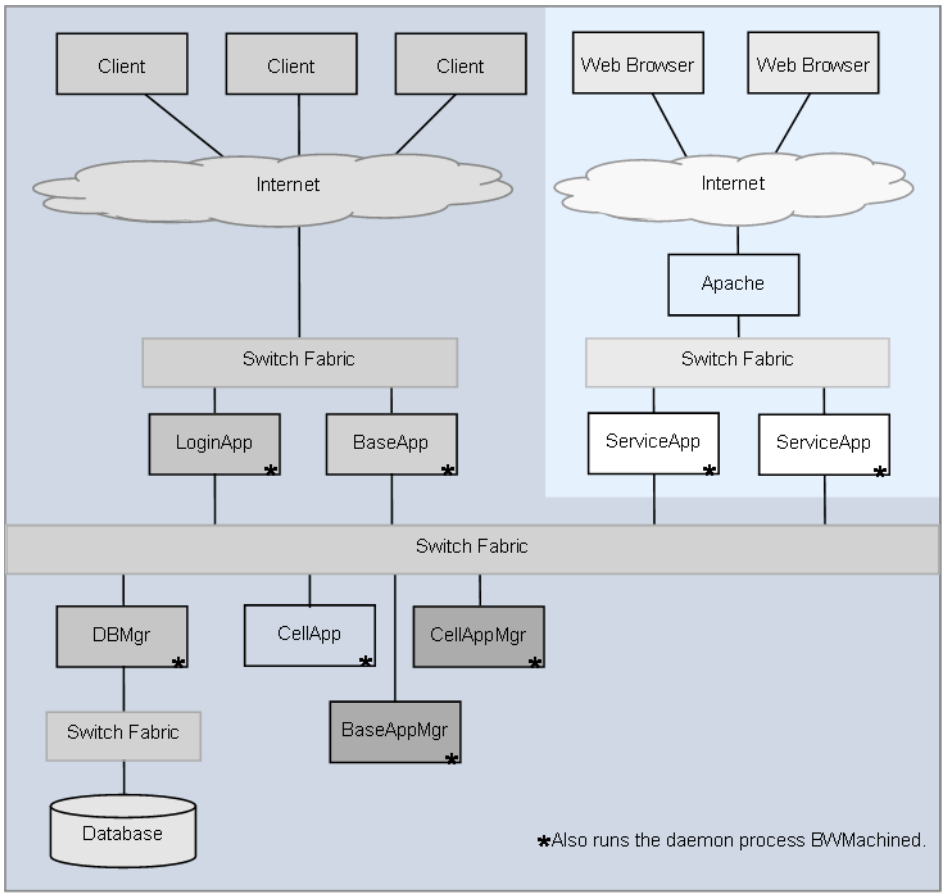
Part II. Using the Web Service from Apache

Table of Contents

Chapter 5. Overview

This section describes a possible method for allowing BigWorld functionality to be integrated into a web server by making use of the web service interface described above. It uses a Linux-based Apache web server setup using PHP (related to the *LAMP architecture*) and interfaces to the BigWorld service via the `Twisted-Web` service. It assumes familiarity with concepts presented in the document Server Programming Guide.

The following diagram shows the standard cluster model, shaded, on the left-hand side, as well as an example of a web integration implementation using an Apache server to provide a web service. For security, ServiceApps will usually not be connected to the internet. Web clients will access the service via the Apache web server, which will communicate with one of the ServiceApps configured to provide the Service. There will usually be multiple ServiceApps providing each Service, for the purposes of load balancing and fault tolerance.



Example configuration of a web service

It should also be noted that while accessing the TwistedWeb service in this way is common, there are many other uses for the TwistedWeb service. Examples include custom administrative tools and statistic gathering scripts.

5.1. 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. 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 6. Configuring Apache

The default web integration implementation uses at its core an Apache http server. The binary for the server is called `httpd`, and it is generally run as a daemon. The binary package and the installation instructions for it can be found at http://apache.org.

Before the server can be accessed, Apache must be told where to find the appropriate server files. This involves putting a symbolic link to the appropriate directory in Apache's `DocumentRoot`, for example `/var/www/html/`. For example, in FantasyDemo, the path to the server files might be `/home/mf/fantasydemo/src/web/php/`, so in order for Apache to be able to find these files, a symbolic link to that path must be placed in `/var/www/html/`. To do this for FantasyDemo, run the following as root:

```
# ln -s /home/mf/fantasydemo/src/web/php/ /var/www/html/fantasydemo
```

The name of the link will be used as the URL segment after the server address, for example, if the link is called `fantasydemo`, as in the command above, then the URL for the server will be `<machine address>/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 target directory, for example `/home/mf/fantasydemo/src/web/php`, and all individual directories in its absolute path. This is likely to require modification only at the top level: the server user's home directory. To grant read access, run the following as root:

```
# chmod o+rx /home/<server-user>
```

SELinux can prevent access to an Apache server if its setting is too strict. To modify the SELinux setting, run the following as root:

```
# system-config-securitylevel
```

From here, set the SELinux level to be no higher than `Permissive`. That is, the setting must not be `Enforcing`.

Restart the Apache server by running the following as root:

```
# /etc/init.d/httpd restart
```

Chapter 7. PHP

As a functional example, 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 sample module is designed to be used under Linux with Apache and mod_php (the PHP module for Apache). The module also works with the PHP Linux command-line interpreter. While the BigWorld sample implementations use PHP, it is possible to implement a web integration system using any such scripting language or web server. As the `TwistedWeb` service services HTTP requests, any mechanism that can perform HTTP requests can integrate with the `TwistedWeb` sample.

7.1. Installation

7.1.1. Installing PHP

You will need to be able to run PHP. It is possible to install it using **yum**:

```
# yum install php
```

You will also need to download and install PHP's libcurl module, which you can also do using **yum**:

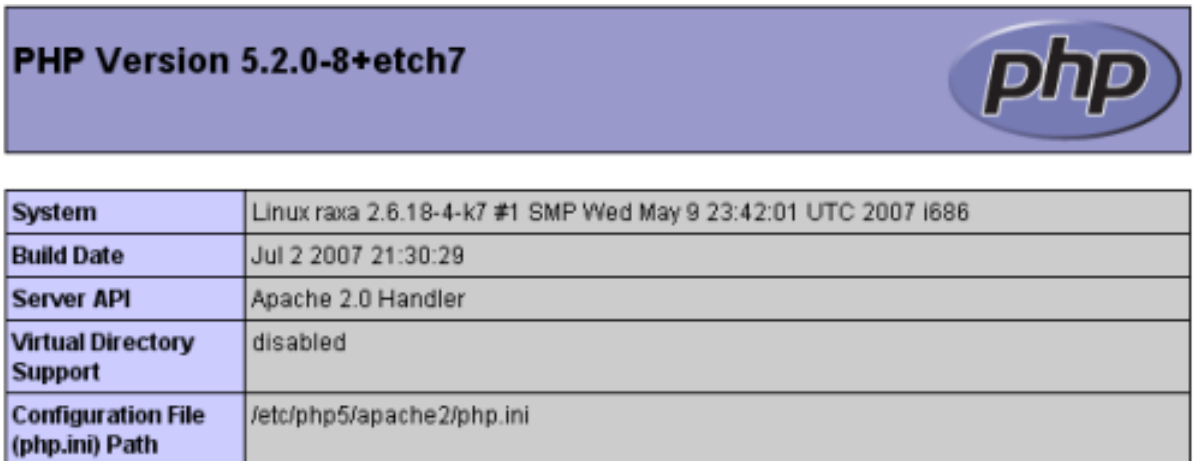
```
# yum install curl
```

7.1.2. Testing

The easiest way to test that PHP is supported is to create a PHP script as illustrated below and use a browser to view it:

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

Testing PHP configuration changes



Example `phpinfo` output

7.2. BigWorld.php

To implement a website that can access the game server, the PHP script needs to make HTTP requests to the TwistedWeb service. To help achieve this, `BigWorld.php` implements a PHP class, called `RemoteEntity`,

that represents a game entity on which you can call methods. This is analogous to a BigWorld server Entity mailbox.

A `RemoteEntity` is created using the start of the URL path as the argument to its constructor. The path does not include the machine name and port or the method to call and method arguments. For more information about Twisted.Web and instructions on how to request entity mailboxes using the TwistedWeb service, see *The TwistedWeb Service* on page 11 . For example, to obtain a `RemoteEntity` to FantasyDemo's `AuctionHouse` global base entity, you would call:

```
$this->auctionHouse = new RemoteEntity( "global_entities/AuctionHouse" );
```

The variable to which the `RemoteEntity` was assigned can subsequently be used like an Entity mailbox for the target entity.

When calling an entity's methods through a `RemoteEntity`, arguments are provided in the form of an array. In PHP, an array is an associated map of key/value pairs. The keys are argument names, and the values are the corresponding argument values. For example, consider the `AuctionHouse` entity's `webCreateBidRangeCriteria()` method, which takes two arguments, representing the minimum and maximum bids for an item, and returns a criteria object. This method has the following definition in `AuctionHouse.def`:

```
<webCreateBidRangeCriteria>
   <Args>
      <minBid>   GOLDPIECES   </minBid>
      <maxBid>   GOLDPIECES   </maxBid>
   </Args>
   <ReturnValues>
      <criteria>   STRING   </criteria>  <!-- Search criteria object, pickled
 -->
   </ReturnValues>
</webCreateBidRangeCriteria>
```

To invoke this method from the `RemoteEntity` for the `AuctionHouse` entity, you will need to create a PHP array of the arguments. The following code illustrates how this would be performed:

```
$res = $this->auctionHouse->webCreateBidRangeCriteria( array(
    "minBid" => $searchMinBid,
    "maxBid" => $searchMaxBid ) );
```

where the values of `$searchMinBid` and `$searchMaxBid` are defined before the call. The return values will be stored in `$res`. In this example, `$res[ "criteria" ]` will contain the string describing the criteria.

The `RemoteEntity` instance converts this call into a HTTP request on an appropriate `TwistedWeb` service fragment. It adds the method name and any arguments. It then blocks waiting for the JSON response. On success, this response is converted into a PHP object.

In addition to calling entity methods on a `RemoteEntity` in order to access the entity on the game server, it is possible to access the same entities directly, by instead using a `RemoteEntity` to access the database, using the TwistedWeb service's `db` URL option, for example:

```
$db = new RemoteEntity( "db" );
```

Using this `RemoteEntity`, it is possible to invoke commands directly on the game server. For example, to directly log an account on to the game server, you could make the following call:

```
$result = $db->logOn( array(
    "username" => $username,
    "password" => $pass ) );
```

7.2.1. BigWorld.php Error handling

It is possible for remote method calls to fail for a number of reasons, including invalid arguments or methods, or the requested entity not existing. When a call on a RemoteEntity instance fails, the TwistedWeb service will return a specially-formatted error object instead of the expected JSON object. As described in “TwistedWeb Error handling” on page 14 , this JSON error object will have the format:

```
{ "excType": "ErrorType", "args": [ "Arg1", "Arg2" ... ] }
```

If such an error object is encountered, `BigWorld.php` will raise it as a PHP exception. This exception type will use the `excType` field as the type name, and the first item in the `args` list as the exception's message:

```
throw new $excType( $args );
```

For example, a JSON object representing a BWInvalidArgsError object will be raised as a BWInvalidArgsError exception, whose message will be the first argument contained in the original JSON object.

Any error object encountered by BigWorld.php will have originated as one of two categories of exception objects:

1. Built-in BigWorld errors - `BWStandardError`

   These are defined in `bigworld/res/scripts/server_common/BWTwoWay.py`, and are explained in the Server Programming Guide's section“BWStandardError”. They originate in the server binaries, and are propagated to the TwistedWeb service.

   In order for them to be raised as exceptions, they are declared as PHP exception objects in `BWError.php`, sharing the class name of their python counterparts.

2. Custom errors - `BWCustomError`

   These are the error types specific to the game scripts, and are described in detail in “PHP Error handling” on page 48. Like the classes derived from `BWStandardError`, these must be declared as php exception objects in order for them to be handled by normal exception-handling procedures. This is to be done in `CustomErrors.php`. For example, if there is a custom error called `MyCustomError` declared in `<res>/scripts/server_common/CustomErrors.py`, there should be a matching php object declared in `CustomErrors.php`:

   ```
   //CustomErrors.php

   <?php
   require_once( "BWError.php" );
   class BWCustomError extends BWFirstArgError {}

   class MyCustomError extends BWCustomError {}
   ?>
   ```

   If an error object is encountered by `BigWorld.php` that has not been declared as its own PHP exception type, it will be thrown instead as a `BWGenericError`, maintaining the `excType` field as part of its exception message:

```
throw new BWGenericError( $excType, $args );
```

For details on handling these errors, refer to “PHP Error handling” on page 48 .

7.2.2. Locating the ServiceApp

The `Service` Singleton class in `BigWorld.php` contains a hard-coded list of possible ServiceApp locations. When a web client starts a new session, the `RemoteEntity` associated with the session is given the mailbox of a random ServiceApp that is currently online and providing a fragment of the desired service. You will need to modify this list to reflect the addresses of your ServiceApp machines:

```
class Service
{
    private function __construct()
    {
        // *** EDIT THIS WITH YOUR ServiceApp ADDRESSES ***
        $this->urlList = array(
            "http://someMachine:8000",
            "http://localHost:8000",
            "http://someOtherMachine:8000"
        );
    }

    ...
}
```

A web client will access a Service through the same ServiceApp for the duration of their session. If this ServiceApp fails during that time, the `RemoteEntity` will find a different ServiceApp providing the same service, and store its address for the remainder of the session.

7.3. RemoteEntity Session Storage

It is possible to store a mailbox to an entity for the duration of a HTTP session. For example, the `BWAuthenticator` class' `authenticateUserPass()` method, mentioned in the previous section, is used not only to invoke the `logOn` command on the game server, but also to hold onto the resulting entity for the duration of the session. By storing the URL of this entity in PHP session variables, whose values are persistent for the entirety of a HTTP session, this allows the web server to require authentication from a web client only once per session.

To store an entity reference:

```
$_SESSION['mailbox'] = "entities_by_id/Avatar/37"; // store details to create
 mailbox later.
```

To later retrieve it:

```
$mailbox = new RemoteEntity( $_SESSION['mailbox'] );
```

For example, to log on and store the result:

```
$db = new RemoteEntity( "db" );

try
```

```
{
  $result = $db->logOn( array( "username" => $username, "password" =>
 $pass ) );
}
catch( BWAuthenticateError $e )
{
  // Handle error
  ...
  return;
}

$_SESSION[ "mailbox" ] = "entities_by_id/" . $result[ "type" ] . '/' .
 $result[ "id" ];
```

Part III. Web Integration Example

Table of Contents

Chapter 8. Overview

This section describes an example web interface to FantasyDemo. It shows how to implement a web auctioning system in order to illustrate functions accessible from the BigWorld web integration module, such as:

- Authentication of player login details.
- Retrieval of an entity mailbox.
- Access to base methods that exist on a base entity through a mailbox.
- Passing parameters to and receiving data from the BigWorld Python scripting layer using the web scripting layer.

The example platform used in this document is the PHP scripting language, combined with the Apache HTTP server. This guide uses a variety of well-documented techniques for web applications (such as handling session variables). Features and techniques used in the example code are common to other web scripting languages; thus the techniques presented can also be used in other web scripting environments.

We provide examples on querying the player for inventory and character statistics. The item and inventory system is based on the BigWorld FantasyDemo item and inventory system. Any item or inventory system with similar concepts of item serial numbers, item types and item locking can be adapted for the web scripts. Other extensions to this model are possible.

We provide a working example of an Auction House, which the player can interact with to:

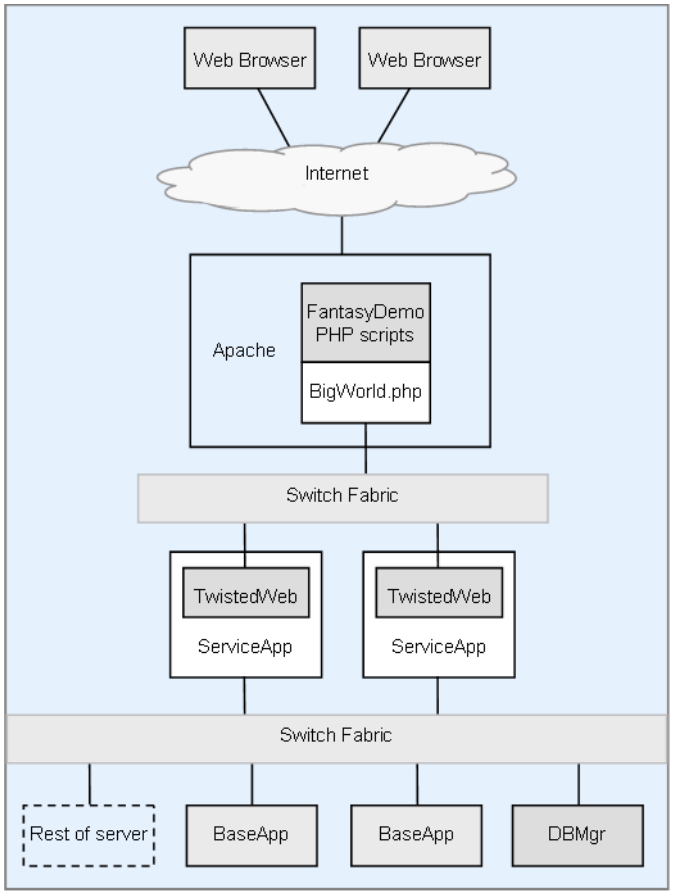
- Create auctions.
- Search for auctions.
- Bid on auctions.

Auctions have the following characteristics:

- Refer to individual items in a player's inventory.
- Have a starting bid, and may optionally have a buyout price.
- Have adjustable expiry times specified when they are created.
- Every player can bid on auctions made by other players.
- Players can specify a maximum value for their bid which allows automated incremental bidding on behalf of the player, up to the specified maximum allowed value.

  This is known as proxy bidding, and this Auction House model is common in other popular Internet-based auction houses. In the code, it is referred to as an `IncrementalAuction`. When entering a maximum bid, the entire amount is considered to be passed to the auction house; on auction resolution, the difference between the maximum bid amount and the actual bid amount is returned back to the player.

We will walk through the source and highlight the salient areas relevant to building a trading system. This example consists of a presentation layer written in PHP, and a logic layer that is implemented as part of the base entity scripts for the appropriate entities, namely `Avatar`, `TradingSupervisor` and `AuctionHouse`.



Block system diagram

The logic for the `AuctionHouse` entity is implemented in the game scripting layer in Python, alongside other entity logic in a BigWorld game. With some alterations, the example code used in this document can be adapted for use in a game that already has a currency and inventory system.

This document assumes that the reader has read the Server Programming Guide, and is familiar with BigWorld Python scripting and has a basic operation of how return-value methods work.

Chapter 9. Use Cases

- The player supplies a username and password to log in, and becomes authenticated against the BigWorld game system — this gives access to character selection, and for a chosen character, character-specific views and operations.
- The player views their character statistics in real-time.
- The player views their inventory and current gold pieces.
- The player nominates an item in their inventory for which he wishes to create an auction, then sets its expiry time, initial bid price, and an optional buyout price.
- The player searches for auctions matching an item type name and/or bid range.
- The player selects an auction and specifies a maximum bid.
- The player logs out.

Chapter 10. PHP Presentation Layer

The presentation layer is written in PHP. It handles requests from web user agents (such as mobile phones and PC browsers) and presents information from the game. The example code PHP sources are found at `fantasydemo/src/web/php`.

10.1. Overview

PHP pages in the example code are represented as PHP objects, and the PHP class definition for `<class>` is located in `<class>.php`. Generally, after the class definition an instance of the page object is created and asked to render itself.

We do not recommend this way of structuring a web interface. The purpose of this PHP construction is to illustrate, as clearly as possible, solutions to common problems encountered when implementing a web interface to a BigWorld game instance. It is not meant to illustrate best practices in web interface implementation.

Developers can use any frameworks that they wish to implement a web interface in PHP or Python. BigWorld does not limit the use of third-party frameworks, from complex systems such as Zope to simple templating engines such as PHP Smarty.

The examples adhere to the XHTML-MP standard (XHTML Mobile Profile).

10.2. Required packages

The Web Integration example requires some packages to be installed on the machine running Apache:

- JSON: TwistedWeb queries return data as JSON objects. Install this package by running the following as root:

```
# yum install php-pecl-json
```

- Image processing: The FantasyDemo PHP scripts make use of the GraphicsDraw library of image processing functions. To install this package, run the following as root:

```
# yum install php-gd
```

10.3. Constants in `Constants.php`

Configuration constants and static data are defined in `Constants.php`. Among other things, it contains:

- The static item type data (such as URLs to image icons, image statistics, etc.) that are used when displaying player inventory.
- The URL of the login page.
- The URL of the welcome page after authentication.

10.4. XHTML-MP helper functions

`XHTML-MP-functions.php` contains functions for commonly used XHTML-MP (XHTML Mobile Profile) element constructs. The simple base ones are listed below:

- `xhtmlMpSingleTag( $name, $className='', $attrString='' )`

Returns the element source of a single unenclosed XHTML element with the given name, class and attribute string.

- `xhtmlMpTag( $name, $contents, $className='', $attrString='' )`

  Returns the element source of a single enclosed XHTML element with the given name, contents, class and attribute string.

- `xhtmlMpAddAttribute( $attrString='', $key, $value )`

  Adds a key value attribute to an attribute string and returns it.

From these, the other common XHTML elements are built. Here are some examples:

- `xhtmlMpHeading( $contents, $level=1, $className='', $attrString='' )`

  Returns the element source of a single unenclosed XHTML element with the given name, class and attribute string.

- `xhtmlMpDiv( $contents, $className='', $attrString='' )`

  Returns the element source for a XHTML `DIV` element with the given optional class and attribute string.

- `xhtmlMpPara( $contents, $className='' $attrString )`

  Returns the element source for a XHTML paragraph.

There is also a `XHTMLMPForm` class for creating XHTML MP forms.

10.5. Debugging PHP example scripts

There is a debug library implemented in `Debug.php` that is used throughout the code example. You can use this to trace the flow of the example scripts using the various debugging output options.

Generally, debug output is displayed in a page as a XHTML comment.

```
class SomePage extends AuthenticatedXHTMLMPPage
{
...
   function renderBody()
   {
      ...
      debug( "this is a test" );
      ...
   }
...
}
```

Example PHP using the debug function

The code above will generate HTTP output like this:

```
<!--
this is a test
-->
```

Example HTTP output

Additionally, you may use this in an overridden `XHTMLMPPage::initialise` method. Because `initialise` does not write output except for HTTP headers in order to perform actions such as HTTP redirects, debug output is deferred until the rendering stage of the page, where you will see debug output as:

```
<!-- deferred error output follows
debug output instance 1
debug output instance 2
debug output instance 3
deferred error output above -->
```

Example HTTP output

There are also some helpful debugging functions for getting representations of more complex PHP objects such as Arrays and class instance objects:

- `debugStringObj`

  Returns the string output.

- `debugObj`

  Sends the string output through `debug()`.

Both these functions generate debug strings representing the objects. This is useful for PHP Arrays and PHP class instance objects.

There is also a registered error handler that prints errors through `debug()`, including information such as stack trace, function line numbers, and passed parameter values.

10.6. XHTMLMPPage objects

These are abstractions of a page, and are the basis for all viewable pages on the web interface. Class definitions can be found in `Page.php`.

There are methods designed to be overridden for the processing stage and the output stage.

The `XHTMLMPPage::initialise()` method is called by `XHTMLMPPage::render()` for processing before any page source is output. Its purpose is to usually set up the page and provide a processing hook for processing HTTP GET/POST request parameters, and initialise page instance variables so they can be easily rendered in the output stage.

`XHTMLMPPage::initialise()` allows you to set redirections from a page to another URL — `XHTMLMPPage::setRedirect()` takes a parameter *`$url`* for this purpose. After calling `initialise()`, if a redirection has been set, then the browser redirects via the HTTP header `Location`.

`XHTMLMPPage::renderBody()` (called from `XHTMLMPPage::render()`) renders the page, and outputs the XHTML element for the page content.

10.7. AuthenticatedXHTMLMPPage objects

Authenticated XHTML Page inherited objects (class `AuthenticatedXHTMLMPPage` in `AuthenticatedPage.php`) are pages that only authenticated users can view. Authentication is performed by an instance of `Authenticator`, with the name of the `Authenticator` class used to do this (which is configured in `Constants.php`). For FantasyDemo scripts, it is the `BWAuthenticator` class.

`Authenticator` objects also provide a means of storing key-value pairs as server-side session variables.

The absence of authentication token variables set in the session indicates that the user is not logged in, which instructs the client browser to redirect to the login page configured in `Constants.php`. This login page

must process requests for logging in so that an authenticator object be created that authenticates the user and their password and creates the necessary authentication token variables.

There is also a timeout for an authenticated session; if no access has been made for a configured amount of time (for details, see `Constants.php`), then the session is invalidated, and browsers that have timed out are redirected back to the configured login page with an error message stating that their session has expired.

Authenticators are used by authenticated pages to check the presence of a valid authentication token:

```
if ($this->auth->doesAuthTokenExist())
{
   $authErr = $this->auth->authenticateSessionToken();
   ...
}
```

10.8. `BWAuthenticator` objects

This class provides an example of how to perform authentication with the BigWorld system. It involves invoking the `db/LogOn` method with the user's name and password:

```
$db = new RemoteEntity( "db" );
$result = $db->logOn( array( "username" => $username, "password" => $pass ) );
```

The result returned is a PHP object containing the entity's type and database id. These details are then stored for later.

```
$this->setEntityDetails( $result[ "type" ], $result[ "id" ] );
```

This stores the string that is required to create a `RemoteEntity`.

```
function setEntityDetails( $type, $id )
{
    ...
    $this->setVariable( BW_AUTHENTICATOR_TOKEN_KEY_ENTITY_PATH,
            "entities_by_id/" . $entityType . '/' . $id );
}
```

The mailbox can then be later retrieved with the `entity()` function.

```
function entity()
{
    $entityPath =
        $this->getVariable( BW_AUTHENTICATOR_TOKEN_KEY_ENTITY_PATH );
    ...

    return new RemoteEntity( $entityPath );
}
```

Authenticated pages can access this mailbox by their authentication object:

```
$playerMailbox = $this->auth->entity();
```

Methods can then be called with:

```
$playerMailbox.someMethod();
```

10.9. Login.php

This page is responsible for collecting the user name and password to be authenticated against the BigWorld server. Thus, any user name and password that is valid when logging in with the FantasyDemo client is also valid here, so that the `bw.xml` configuration options `dbMgr/createUnknown` and `dbMgr/rememberUnknown` become relevant (for details on these configuration options, see the document Server Operations Guide's section *Server Configuration with `bw.xml`* → "DBMgr Configuration Options").

Authentication is performed by making a request to the authenticator object, for example:

```
$this->auth->authenticateUserPass( $_REQUEST['username'],
 $_REQUEST['password'] );
```

10.10. Characters.php

Once the user authenticates using a username and password, the `Account` mailbox is queried for its list of associated `Avatar` characters. This is done as follows:

```
$account = $this->auth->entity();
$res = $account->webGetCharacterList();
```

This returns the list of character descriptors in `$res['characters']`. Each character descriptor is a dictionary with keys `name` and `type` (of entity, usually `Avatar`).

You can also create characters via this page:

```
$res = $account->webCreateCharacter( array( "name" =>
 $_GET['new_character_name'] ) );
```

You choose a character to progress. Once chosen, the session player `Account` mailbox is replaced by a mailbox to the player `Avatar` entity and the keep-alive period is set on the newly made character mailbox.

```
$res = $account->webChooseCharacter( array(
            "name" => $_GET[ 'character' ],
            "type" => "Avatar" ) );
...
$this->auth->setEntityDetails( $res[ "type" ], $res[ "id" ] );
```

10.11. News.php

This page is the entry point after a user has logged in and chosen a character. Currently, this is a static PHP page, but one possible extension to this is to have a `News` entity in the world, which is queried by this page each time it loads up.

A hook for doing this is present in the `NewsPage::initialise()` method:

```
// the articles could also come from an entity
```

```
// e.g.
// $newsagent = new RemoteEntity( "global_entities/NewsAgent" );
// $res = $newsagent->getNewsArticles();
// $this->articles = $res['articles'];
```

10.12. Character.php

This page queries the player `Avatar` mailbox in real-time via the `Avatar.webGetPlayerInfo()` web method for the current statistics of the player for display:

```
$player = $this->auth->entity();
$res = $player->webGetPlayerInfo();
```

The position and the direction the player is currently facing is also reported back through this page if the player is online.

10.13. Inventory.php

This page queries the player character mailbox via the `Avatar.webGetGoldAndInventory()` web method. This method is defined in the entity definitions file for the `Avatar` entity type, in `fantasydemo/res/scripts/entity_defs/Avatar.def`:

```
<webGetGoldAndInventory>
   <ReturnValues>
      <!-- The Avatar's available gold pieces. -->
      <goldPieces>        GOLDPIECES  </goldPieces>

      <!-- List of item descriptions as dictionaries with keys:
           'serial': the serial number of the item
           'itemType': the item type
           'lockHandle' : lock handle associated with this item
      -->
      <inventoryItems>    PYTHON      </inventoryItems>

      <!-- List of dictionary with keys:
          'serials': a list of serial numbers of locked items
          'goldPieces': the gold pieces locked
      -->
      <lockedItems>       PYTHON      </lockedItems>
    </ReturnValues>
</webGetGoldAndInventory>
```

The excerpt above shows that that the return value to the PHP scripting layer is an Array with keys `goldPieces`, `inventoryItems` and `lockedItems`. We store them in the class instance variable `$this->inventory`:

```
$entity =& $this->auth->entity();
...
$this->inventory = $entity->webGetGoldAndInventory();
```

The gold pieces are accessible from this instance variable when displaying its value to the user:

```
echo(
   xhtmlMpDiv(
      'Gold: '. $this->inventory['goldPieces'],
```

```
            'goldRow'
    )
);
```

This page is also responsible for handling requests for creating auctions from the player. The player nominates an item in their inventory, based on its serial number, then sets the initial auction parameters through the form and on submission, and creating the auction is a case of invoking the `Avatar.webCreateAuction`:

```
$res = $entity->webCreateAuction( array(
    "itemSerial" => $itemSerialToAuction,
    "expiry" => $expiry,
    "startBid" => $bidPrice,
    "buyout" => $buyout ) );
```

10.14. PlayerAuctions.php

This page enables players to see the state of auctions they have created. The search criteria used is an instance of `SellerCriteria` with the current player's database ID. This is retrieved from `Avatar.webGetPlayerInfo()`:

```
$res = $this->player->webGetPlayerInfo();
$this->playerDBID = $res['databaseID'];
```

The result is used in constructing and applying the `SellerCriteria` for getting search results to present to the user.

10.15. SearchAuctions.php

This page enables players to search for auctions by using the singleton `AuctionHouse` entity, and allows for bidding of searched auctions.

Search criteria objects are built up using various methods in AuctionHouse, for example:

```
$res = $this->auctionHouse->webCreateItemTypeCriteria( array(
    "itemTypes" => $itemTypesList ) );
...
$searchCriteria = $res['criteria'];
...
$res = $this->auctionHouse->webCreateBidRangeCriteria( array(
    "minBid" => $searchMinBid,
    "maxBid" => $searchMaxBid ) );
$bidRangeCriteria = $res['criteria'];
$res = $this->auctionHouse->webCombineAnd( array(
                    "criteria1" => $searchCriteria,
                    "criteria2" => $bidRangeCriteria ) );
$searchCriteria = $res['criteria'];
```

The `$searchCriteria` object contains the combined search criteria. It can be applied to a search via the `AuctionHouse.webSearchAuctions()` method. This method returns a list of auction IDs that match the criteria.

To retrieve the auction descriptors (which contains information such as the seller player, the current bid amount, the item type), we use the `AuctionHouse.webGetAuctionInfo()` which takes a list of auction IDs and returns a list of auction descriptors.

```
$res = $this->auctionHouse->webSearchAuctions( array(
            "criteria" => $searchCriteria ) );
...
$res = $this->auctionHouse->webGetAuctionInfo( array(
            "auctions" => $res["searchedAuctions"] ) );
...
// store the auctions in an associative array by auction ID
$this->searchResults = Array();
foreach ($res['auctionInfo'] ) as $auctionInfo)
{
    $this->searchResults[$auctionInfo['auctionID'] = $auctionInfo;
}
...
```

10.16. PHP Error handling

As described in section "BigWorld.php Error handling" on page 29 , all mailbox queries can fail, resulting in an error object being returned. In addition to errors coming from the server binaries, errors can be returned from the remote methods themselves. These are custom exception types that will be raised by the auction house scripts, the authentication scripts, and the other methods called remotely by the web client. For example, in the AuctionHouse methods called by the FantasyDemo web client PHP code, there are such error classes as `InsufficientGoldError` and `SearchCriteriaError`, allowing error messages that are displayed to the player to be as helpful as possible.

The file `fantasydemo/res/scripts/server_common/CustomErrors.py` contains the declarations of these custom Python exception classes. They are as follows:

- `AuctionHouseError`

  Attempt to access an AuctionHouse entity that is not allowed or non-existent

- `BidError`

  Bid for an auction with an invalid amount

- `BuyoutError`

  Set an invalid buyout price for an auction

- `CreateEntityError`

  An entity can't be created

- `DBError`

  A database query or modification fails

- `InsufficientGoldError`

  The player doesn't have enough gold for the desired transaction

- `InvalidAuctionError`

  Attempt to access an invalid or non-existent auction

- `InvalidDamageAmountError`

  Attempt to deal an invalid amount of damage to an entity

- `InvalidItemError`

An item can't be accessed

- `ItemLockError`

  An item can't be locked

- `PriceError`

  Set an invalid price for an auction

- `SearchCriteriaError`

  Search criteria for an auction are invalid

The TwistedWeb service will catch these exception objects, convert them to JSON objects and return them instead of the queried response object, as explained in “TwistedWeb Error handling” on page 14 .

Additional error types can be used by declaring them in `CustomErrors.py`. They must be derived from `BWTwoWay.BWCustomError`. They should also be declared in `CustomErrors.php`, to allow them to be handled individually. For example:

```
# CustomErrors.py

from BWTwoWay import BWCustomError

class MyCustomGameError( BWCustomError ):
    pass
```

```
// CustomErrors.php

require_once( "BWError.php" );

class BWCustomError extends BWError {}

class MyCustomGameError( BWCustomError ):
    pass
```

After `BigWorld.php` receives a response, it decodes the returned JSON object. If the object is found to be an error, it will be raised as a PHP exception. The caller of the mailbox query must therefore handle any potential exceptions that may be raised. For example:

```
try
{
    $result = $mailbox->someMethod();
}
catch( Exception $e )
{
    addExceptionMsg( $e );
    return;
}

// Use $result
...
```

All built-in BigWorld errors and custom errors extend a common base class, `BWError`. This class implements a `message` method, which creates a string containing both the underlying error type, and the exception

message. It also returns a well-formatted string for BWGenericError objects, described in “BigWorld.php Error handling” on page 29 . This method can be used to generate error messages in an exception handler. For example:

```
function getExceptionMsg( $exception )
{
    if ($exception instanceof BWError)
    {
        return $exception->message();
    }
    else
    {
        // uses the built-in getMessage method
        return $exception->getMessage();
    }
}

function addExceptionMsg( $exception )
{
    $msg = $this->getExceptionMsg( $exception )

    // Create an error message from $msg
    ...
}
```

For example:

```
throw new NoConnectionError( "Unable to contact game server" );
```

If the above exception is caught and the object sent as the argument to addExceptionMsg, the player will receive the following error message:

```
NoConnectionError: Unable to contact game server
```

Alternatively, if an instance of a `BWGenericError` or a non-`BWError` object is thrown with the same message, only that message would be emitted, and not the exception type:

```
Unable to contact game server
```

The complete error handling mechanism for the FantasyDemo web client is implemented in `Page.php`.

`CustomErrors.php` also defines an error class for errors specific to the PHP, called `BWPHPError`. Errors of this type can be used to take advantage of the formatting of error messages using the `message` method provided by `BWError`. This allows PHP errors to be handled and reported in a manner consistent with the error objects encountered by `BigWorld.php`:

```
// CustomErrors.php

class BWPHPError extends BWError {}

class InvalidFieldError extends BWPHPError {}
```

```
// BWAuthenticator.php
```

```
...
function authenticateUserPass( $username, $pass )
{
    if ($username == '')
    {
        throw new InvalidFieldError( "Username is empty" );
    }
    if ($pass == '')
    {
        throw new InvalidFieldError( "Password is empty" );
    }
    ...
}
...
```

```
// Login.php

...
    try
    {
        $auth->authenticateUserPass( $username, $pass );
    }
    catch( InvalidFieldError $e )
    {
        addExceptionMsg( $e );
        return;
    }
...
```

Attempting to log in with an empty username will result in the following error message being displayed:

```
InvalidFieldError: Username is empty
```