How To Implement Web Integration

BigWorld Technology 2.0. Released 2010.

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 to implement a web interface to a running BigWorld game cluster, and 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 from the web scripting layer.

For details on the integration of BigWorld functionality into a web server, see the document BigWorld Web Integration Reference.

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, such as mod_python.

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:

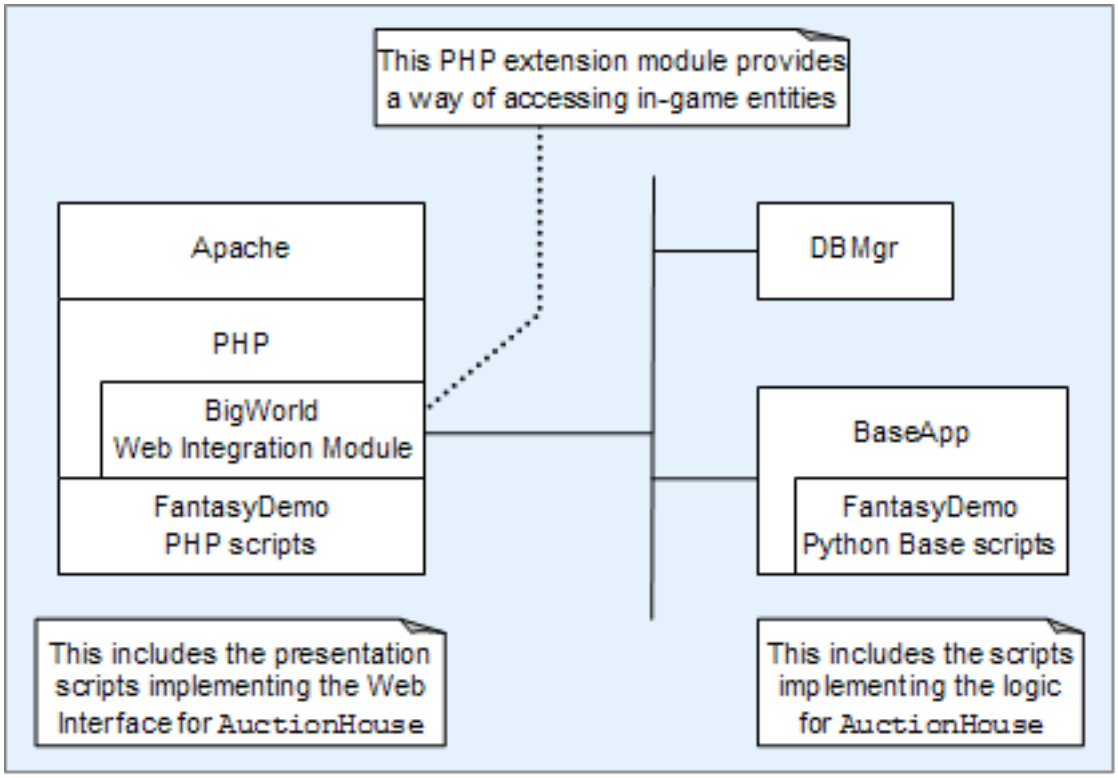
- Create auctions.
- Search for auctions.
- Bid on auctions.

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 — the system will automatically bid in increments on behalf of the player, up to the specified maximum.

  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 BigWorld Web Integration Reference, and is familiar with BigWorld Python scripting and has a basic operation of how return-value methods work.

1.1. Build and configuration

The Python web extension and the PHP web extension are both required to be built — for details on how to do that, see the document BigWorld Web Integration Reference's sections *Python* → "Building the Module From Source" and sections *PHP* → "Building the Module From Source".

The PHP web extension provides a compatibility layer for PHP to use the functions present in the Python web extension.

Chapter 2. 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 3. BigWorld Python Scripting Layer

There are three entity types that are involved in an auction transaction:

- The player `Avatar`.
- The `AuctionHouse` entity.
- The `TradingSupervisor`.

For details on how to implement safe atomic trading transactions between two entities instantaneously, see the document How To Implement Items And Trading.

The Auction House example code can be traced by following the `AuctionHouseCommon` debug output — if the `AuctionHouseCommon.DEBUG` module attribute is set to `True`, then those statements are output to the BaseApp console. You can set this by editing the `fantasydemo/res/scripts/common/AuctionHouseCommon.py` Python module file.

3.1. Item and inventory system

In FantasyDemo, items have a numerical item type and an entity-wide serial number (*i.e.* the serial number is specific to that entity), and can be locked under a lock handle. For details on how items are implemented in FantasyDemo, see the document How To Implement Items And Trading.

In FantasyDemo, when an item is locked, it is un-equipped on the cell if the player cell entity exists and it has that item equipped, and it cannot be used by the client entity until unlocked. Gold pieces can also be locked in conjunction with a set of items.

The example code for the Auction House system is loosely tied to this scheme. Items are passed loosely as Python tuples, and appropriate callbacks and hooks are used that can be replaced. With relatively minor modifications, developers can use the example code to provide their own inventory system, and use the item locking hook to do any pre-transactional checks and actions.

3.2. Retrieving real-time player data

In FantasyDemo, there are two scenarios when a player log on to the web interface: either they have a cell entity, they are running around the world, or they are not instantiated on any cell, and only the Base entity exists on the server.

In FantasyDemo, the cell data that we expose to the web interface consists of:

- The player's name.
- The player's health and maximum health.
- The frags, or the number of kills made by the player.
- Position vector and direction vector of the player.

When there is no cell entity present, the values of the player name health, maximum health, and frags exist in a dictionary on the base called `cellData`. However, when there is a cell entity present, the values are required to be queried from the cell — via the `Avatar.getPlayerInfoBaseRequest()` cell method and the `Avatar.onPlayerInfo()` callback method on the base.

3.3. Auction house data types, persistence and aliases

There are some data type aliases specific to the Auction House system which are defined in `fantasydemo/res/scripts/entity_defs/alias.xml`:

- AUCTIONID
- AUCTIONEXPIRY
- AUCTIONBIDDER
- AUCTIONBIDDERS
- AUCTION
- AUCTIONMAPPING
- AUCTIONS
- AUCTIONINFO

For details on the entity definitions alias file and advanced persistence of user-defined data types, see the document the Server Programming Guide 's section the “Alias of Data Types”.

Notice that the AUCTIONID data type is aliased to a STRING. In the example code, the AUCTIONID is comprised of the following string format:

```
<player entity database ID>/<BigWorld game time>
```

For example:

```
432/456.23
```

IncrementalAuction objects are defined in Python in the AuctionHouseCommon module, and are database-persistent. The corresponding AUCTION data type is aliased to a class-customised FIXED_DICT type; the Python scripts for streaming and de-streaming Auction objects to and from the database are in the AuctionDataTypes module located in fantasydemo/res/common, in the class AuctionDataType.

Items in IncrementalAuction are represented as a 3-tuple, as displayed below:

```
( lockHandle, itemType, itemSerial )
```

There also exists a collections data type for holding a map of auction IDs to auction objects. This is the type of the auctions base property on the AuctionHouse entity that holds all auctions specific to that auction house.

The AUCTIONS data type manifests itself as a dictionary from auction IDs to IncrementalAuction objects to the Python scripts. The corresponding AUCTIONS entity data type is aliased to a class-customised FIXED_DICT implemented by a class in the AuctionDataTypes module, in the class AuctionsDataType. It consists of an array of AUCTIONMAPPING, which is aliased to an non-customised FIXED_DICT data type with auction ID and auction object properties. These alias definitions are present in fantasydemo/res/scripts/entity_defs/alias.xml.

3.4. Auction life cycle and logic

The life cycle of an auction follows the steps described below:

1. Auctions are created by player Avatar entities by passing to Avatar.webCreateAuction() the serial number of the item that they wish to sell along with the auction parameters, such as expiry, the start bid price, and the optional buyout price.

The player `Avatar` entity locks the item and receives a lock handle.

2. The player `Avatar` entity makes a request to the `AuctionHouse` entity via `AuctionHouse.createAuction()` to create an auction with the item's type, serial and lock handle in a 3-tuple, and the auction parameters.

   The `Avatar` entity also supplies itself as a mailbox so that it can be called back on when the auction has been registered, or if there was an error — in case of error, the `Avatar` entity can unlock the item.

3. The `AuctionHouse` entity registers the new auction, and makes it available for searching by other players by adding it to its `AuctionHouse.auctions` collection member property.

   It also sets the auction's expiry time to after the specified expiry period after the current `BigWorld.time()`, then writes itself to the database.

4. Other players can search for auctions by creating criteria (for details, see "Searching and search criteria" on page 14) and applying them to `AuctionHouse.webSearchAuctions()`, which returns an array of auction IDs, rather than auction descriptors (which are returned by `AuctionHouse.webGetAuctionInfo()`).

   This is to support implementations of paging through search results on the web interface.

5. The bidder `Avatar` entity locks the maximum bid amount once a bidder player has selected an auction and has bid on it through the web interface (via `Avatar.webBidOnAuction()`, by supplying the auction ID and the maximum bid amount). If this fails due to insufficient funds:

   - Control is returned to the web scripting layer.
   - The bidding player is notified of failure due to insufficient funds.
   - No further action or change is taken.

6. If the bid is successful, the lock handle is passed to the `AuctionHouse` entity through the `AuctionHouse.bidOnAuction()` method to register the bid with the auction ID, bidder player mailbox, and the bidder's maximum bid amount lock handle.

7. If the auction has no current bidder, then the bidder entity becomes the new highest bidder, and the bidding price is incremented once. Otherwise, in the case of a bidder bidding against an existing highest bidder, the auction bidding is done incrementally up to the minimum of the maximum bids of the two competing bidders.

8. If the bid is successful, then this bidder becomes the new highest bidder, and the bid price may increment up to the amount of the maximum bid before this bidder becomes outbid.

   If this bidder outbids another bidder, then that bidder's entity is found, and has the method `Avatar.onAuctionOutbid()` called on it to notify that `Avatar` entity that it has been outbid, which unlocks their maximum bid's lock handle, effectively returning the gold back to the outbid bidder player.

   Otherwise, if the bid has failed, then the contesting bidder has `Avatar.onAuctionOutbid()` called on its entity, which unlocks the lock handle to return the gold back to the contesting bidder player.

9. Auctions are checked periodically with a BigWorld timer that is set through `Base.addTimer()` every `AuctionHouse.AUCTION_CHECK_PERIOD` (a Python module-level property) seconds.

   When an auction expires, if at the time of expiry, it does not have a bidder registered:

   - The auction is considered expired and is removed.
   - The seller entity has `Avatar.onAuctionExpired()` called back on.

- The item is unlocked.

If it has a bidder at the time of expiry:

- The auction is won by the current bidder.
- The item locked by the seller must be exchanged by the amount of gold pieces that the auction has been won at. This can be lower than the maximum bid amount entered by the highest bidder, and so there is a mechanism to relock a lower amount of gold in order to complete the auction transaction.

  The `Avatar.lockAuctionGold()` accomplishes this by being passed from the `AuctionHouse` entity:

  - The auction ID.
  - The actual amount of gold pieces that the auction was won at.
  - The pre-existing lock handle for the maximum bid amount, in gold pieces.
- This causes the old lock on the maximum bid amount to be unlocked, and since the actual amount that the auction expired at is always less than or equal to the maximum bid amount, this is guaranteed to succeed.

  This auction bid amount at expiry is relocked, and is passed back to the `AuctionHouse` entity through its `AuctionHouse.completeAuction()` method, along with this player's mailbox and the auction ID.
- The `AuctionHouse` entity locates the seller entity and the bidder entity for exchanging locked items.

  The actual exchange is carried out by re-using functionality from the FantasyDemo `TradingSupervisor` entity, which handles atomic exchanges between two `Avatar` entities and locks on items held by the respective entities (for details on how the `TradingSupervisor` exchanges items, see the document How To Implement Items And Trading's section *Trading* → "Trading Supervisor").
- The auction is removed.

3.5. Web method conventions

The convention used in the base entity scripts for implementing web methods is to prefix each method by the string `web` (for example, `webGetPlayerInfo()`). Please note that this is only a convention used in our example code, and it is not a requirement.

Many of the web return-value methods also have return values for returning the status of the operation back to the web scripting layer:

- *`success`*::BOOL — Whether the operation succeeded.
- *`errMsg`*::STRING — Error message string, describing the reason for the failure.

3.6. Callbacks

There are some callback methods that are invoked by the `AuctionHouse` entity to the `Avatar` entity in response to `AuctionHouse` requests, for creating auctions, bidding on, and buying out auctions.

In the `Avatar` entity scripts, there is a Python property set at `__init__()` time called `tradingCallbacks`. This is a map of callback names to maps of unique identifiers (depending on what kind of call it is) to callbacks implemented by `functools.partial` instances and arguments. This is partly to avoid sending unnecessary context across to the `AuctionHouse` entity by preserving it in the memory space of the `Avatar`

entity and have that context restored when the `AuctionHouse` calls back. The `Avatar` entity de-multiplexes on these callbacks based on a unique key for that set of transactions.

3.7. The Auction House abstraction

In the AuctionHouseCommon module, there are some non-BigWorld-specific helper classes that are intended to be sub-classed and functionality to be overwritten. These classes cover the underlying logic of auctions and auction houses, but are not tied to any persistence mechanism. There is a sample of an `InMemoryAuctionHouse`, which maintains lists of auctions in memory. Different auction types can be added. Different buyers and seller objects can be used, as long as they adhere to the protocol.

The abstraction consists of:

- `AbstractAuctionHouse`
  - Handles requests for searching auctions.
  - Auction administration and persistence.
- `Auction`
  - Handles the auction mechanics of bidding.
  - Holds a reference to the seller interface instance.
  - Holds a reference to the item object.
  - Holds a reference to the bidder interface instance, if one exists.
  - Holds a reference to the auction house object.
  - Holds the auction data, such as expiry, bidding price, etc.
- `Seller`
  - Auction seller interface.
  - Hook for client code to receive auction house events for sellers.
- `Bidder`
  - Auction bidder interface.
  - Hook for client code to receive auction house events for bidders.
- `AuctionSearchCriteria`
  - Definition of the protocol for search criteria to be used when filtering auctions for a search query.

Some generic sub-classes are present for these abstractions:

- `AuctionHouse`

  Entity that is a BigWorld implementation of `AbstractAuctionHouse`, using the BigWorld persistence engine.

- `FantasyDemoSeller`

  Implementation of the `Seller` interface — this sub-class holds the database ID of the seller `Avatar` entity.

- `FantasyDemoBidder`

Implementation of the `Bidder` interface — this sub-class holds the database ID of the bidder `Avatar` entity, as well as the lock handle for the bidder's maximum bid.

3.8. Searching and search criteria

In order to support arbitrary Boolean operations on auctions, search criteria objects are created to filter matching auctions. They are supported on the BigWorld server side by Python objects of type `AuctionSearchCriteria`, and passed to the web scripting layer as Python pickled strings of those criteria objects. The web scripting layer constructs instances of these criteria objects, and can store instances of them as Python pickled strings — they pass them back to the `AuctionHouse` entity when retrieving auction search results.

The `AuctionSearchCriteria` class has one method to be overridden by sub-classes: the `filter()` method:

```
class SomeSearchCriteria( AuctionHouseCommon.AuctionSearchCriteria ):
   def filter( self, auction ):
      """
      Return True if the given auction satisfies this criteria, otherwise
 False.
      """

      # test the auction and return either True or False
      return ...
```

There are a few criteria types already defined. The definitions for these classes can be found in:

- `fantasydemo/res/scripts/common/AuctionHouseCommon.py`
- `fantasydemo/res/scripts/base/AuctionHouse.py`

The web return-value method declarations can be found in the entity definitions file `fantasydemo/res/scripts/entity_defs/AuctionHouse.def`.

It is fairly easy to add new filtering criteria by implementing the interface presented in `AuctionSearchCriteria` for other item and inventory subsystems:

- `ItemTypeCriteria`

  Search for auctions with an item type in the list of given item types. This is specific to FantasyDemo, but a similar criteria for matching items in other item and inventory systems could also be written.

  Created by the base return-value method `webCreateItemTypeCriteria()`.

- `BidRangeSearchCriteria`

  Search for auctions that satisfy a given bid range.

  Created by the base return-value method `webCreateBidRangeCriteria()`, taking as arguments the minimum and maximum in the bid range (either argument is optional, but not both).

- `BidderCriteria`

  Search for auctions of the bidder identified by the specified database ID.

  Created by the base return-value method `webCreateBidderCriteria()`.

- `SellerCriteria`

Search for auctions of the seller identified by the specified database ID.

Created by the base return-value method `webCreateSellerCriteria()`.

- `AndSearchCriteria`, `OrSearchCriteria`

  Composite criteria classes available for chaining criteria together in arbitrary ways.

  These criteria can be accessed through the `AuctionHouse` base return-value methods `webCombineAnd()` and `webCombineOr()`, and providing each with two pickled criteria objects to be combined using the given Boolean operation.

  Other pickled instances of `OrSearchCriteria` and `AndSearchCriteria` can also be supplied.

3.9. `Avatar` entity

The web interface scripts deal with the Avatar entity (the player's entity) when doing player-specific actions, such as querying player inventory and statistics, or bidding using a certain item in the player's inventory.

The `Avatar` base entity implements return-value methods meant to be accessible from the web for returning inventory information, querying player data, and executing requests for bidding on other players' auctions. These are:

- `Avatar.webGetPlayerInfo()`

  Retrieves information about the player, such as:

  - Database ID.
  - Player name.
  - Online status.
  - Position in the world, if they are online.
  - Direction facing in the world, if they are online.
  - Health.
  - Maximum health.
  - Number of frags.

  If the player is online, then the information is collected from the cell entity, via the cell method `Avatar.getPlayerInfoBaseRequest()`, which returns the information via the base method `Avatar.onPlayerInfo()`.

- `Avatar.webGetGoldAndInventory()`

  Retrieves the available gold pieces and the state of the inventory. The inventory items are returned as a list of dictionaries with information about the type, serial number and lock state of each item in the player's inventory.

- `Avatar.webCreateAuction()`

  Locks an item in the inventory identified by serial number, and creates an auction on the `AuctionHouse` singleton entity with the given auction parameters (*e.g.*, expiry, starting bid, optional buyout amount).

- `Avatar.webBidOnAuction()`

  Lodges a maximum bid on, or to buyout, an auction with the given auction ID.

There are callbacks used to notify the player `Avatar` entity about auction house events:

- `Avatar.onCreateAuction()`

  Called in response to creating a new auction by the player.

- `Avatar.onBidOnAuction()`

  Called in response to bidding on another player's auction.

- `Avatar.onBuyoutAuction()`

  Called in response to buying out another player's auction.

- `Avatar.onAuctionWon()`

  Called when an auction has been won by this player.

- `Avatar.onAuctionOutbid()`

  Called when an auction has been outbid on by another player.

- `Avatar.onAuctionExpired()`

  Called when an auction created by this player has expired without any other player bidding on it.

In addition, the `Avatar.lockAuctionGold()` method is used by the `AuctionHouse` entity for auction resolution. It relocks a lower bid amount of gold pieces for an auction at the bid price that the auction was won at. Also supplied is an existing lock handle for the maximum bid amount committed to by a bidder; the difference between the maximum bid and the bid amount at time of auction expiry is returned to the player this way. `AuctionHouse.completeAuction()` is called back from `Avatar.lockAuctionGold()` when the new lock has been created.

There are also two methods used for querying character statistics, some of which exist on the cell, from the base:

- `Avatar.getPlayerInfoBaseRequest()`

  Cell method called by the base to request character statistics from the cell.

- `Avatar.onPlayerInfo()`

  Base method called back by the cell to supply character statistics from the cell.

The actual locked item swapping mechanism uses the methods:

- `Avatar.tradeCommitActive()`
- `Avatar.tradeCommitPassive()`

For details on how the swapping works via these methods, see the document How To Implement Items And Trading's section *Trading* → "Trading Supervisor").

3.10. `AuctionHouse` entity

`AuctionHouse` is a subclass of `BigWorld.Base` and `AuctionHouseCommon.AbstractAuctionHouse`, and is responsible for:

- Holding auctions.
- Bidding on auctions.

- Checking auction expiries.
- Responding to requests from `Avatar` entities for creating auctions.
- Responding to requests for searching through auctions.

This is a base-only singleton entity — uniqueness is controlled through looking up global bases at `__init__()` time and self-destructing if a global base called `AuctionHouse` already exists. On BaseApp start-up, `AuctionHouse.wakeup` is called to load the `AuctionHouse` entity if it does not already exist (for details, see `FantasyDemo.onBaseAppReady()` in the FantasyDemo personality script `fantasydemo/res/scripts/base/FantasyDemo.py`).

The entity definitions file for this entity can be located at `fantasydemo/res/scripts/entity_defs/AuctionHouse.def`.

The base script for the `AuctionHouse` entity can be located at `fantasydemo/res/scripts/base/AuctionHouse.py`.

The `AuctionHouse` base entity contains the base property auctions which is an instance of the `AUCTIONS` collection data type (for details on how the different data types used in the auction house system interact, see “Auction house data types, persistence and aliases” on page 9) .

The `AuctionHouseCommon` module is imported by `AuctionDataTypes` in order to support database persistence of auction house data. For details, see “Auction house data types, persistence and aliases” on page 9 and “The Auction House abstraction” on page 13).

The `AuctionHouse` entity exposes the following methods to the web interface scripting layer:

- `AuctionHouse.webGetTime()`

  Retrieves the `BigWorld.time` value to the web scripting layer.

- `AuctionHouse.webSearchAuctions()`

  Searches for auctions that match the given criteria object, and returns only the auction IDs.

- `AuctionHouse.webGetAuctionInfo()`

  Returns information about the given auction IDs in the form of a dictionary of properties.

- `AuctionHouse.webCombineAnd()`

  Combines two criteria objects together using the `AND` Boolean function.

- `AuctionHouse.webCombineOr()`

  Combines two criteria objects together using the `OR` Boolean function.

- `AuctionHouse.webCreateItemTypeCriteria()`

  Creates a criteria that matches auctions based on item types.

- `AuctionHouse.webCreateBidRangeCriteria()`

  Creates a criteria that matches auctions base on the bid price. Ranges can be open (*i.e.*, either the top or the bottom limits can be omitted).

- `AuctionHouse.webCreateSellerCriteria()`

  Creates a criteria that matches auctions based on the seller's database ID.

- `AuctionHouse.webCreateBidderCriteria()`

  Creates a criteria that matches auctions based on the bidder's database ID.

3.11. `TradingSupervisor` entity

This entity oversees and facilitates in-game trading and atomic swap transactions. For details on this entity's design and implementation, see the document How To Implement Items And Trading's section *Trading* → "Trading Supervisor".

Chapter 4. PHP Presentation Layer

The presentation layer is written in PHP, and 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`.

4.1. 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).

4.2. Device-specific handling and WURFL

Since there is a large variability in the device capabilities of each user agent, we present a solution for handling multiple device types using WURFL.

WURFL is a database of user agents and their capabilities, and given the user agent string, we can access the device's capabilities (such as screen resolution) and adjust our presentation accordingly. WURFL has script support in PHP and Python (among other languages). The latest release of WURFL can be downloaded from http://wurfl.sourceforge.net.

WURFL requires some initial set-up before it can be used in PHP; in particular, the device capabilities cache needs to be created. For details on how to install and update the cache, see the WURFL documentation.

4.3. 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.

4.4. 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.

4.5. 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.

4.6. XHTMLMPPage objects

These are abstractions of a page, and are the basis for all viewable pages on the web interface — their definition is 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.

4.7. 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();
   ...
}
```

4.8. `BWAuthenticator` objects

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

```
$logonResult = bw_logon( $username, $pass,
    /* allow_already_logged_on */ TRUE );
```

The result returned by `bw_logon()` is `TRUE` if it succeeded, or an error message with the reason why the logon operation failed.

```
$mailbox = bw_look_up_entity_by_name( "Account", $username );
```

We then retrieve the mailbox by calling `bw_look_up_entity_by_name()`. We know it is of the `Account` type because that is the initial entity type when a user logs in (this is defined in `bw.xml` option `dbMgr/entityType`).

```
bw_set_keep_alive_seconds( $mailbox, AUTHENTICATOR_INACTIVITY_PERIOD );
```

A keep-alive period is set on this mailbox after we have retrieved it.

```
function &getMailbox()
{
   $mailboxSerialised =& $this->getVariable(
      BW_AUTHENTICATOR_TOKEN_KEY_MAILBOX );
   ...
   return bw_deserialise( $mailboxSerialised );
}
```

Authenticated pages can access this mailbox by their authentication object:

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

4.9. `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'] );
```

4.10. 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:

```
$player = $this->auth->getMailbox();
$res = bw_exec( $player, "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 = bw_exec( $player, "webCreateCharacter", $_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 = bw_exec( $player, "webChooseCharacter", $_GET['character'], "Avatar" );
$mailbox = $res['character'];
bw_set_keep_alive_seconds( $mailbox, AUTHENTICATOR_INACTIVITY_PERIOD );
```

4.11. 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 = bw_lookup_entity_by_name( "NewsAgent", "NewsAgent" );
// $res = bw_exec( $newsagent, "getNewsArticles" );
// $this->articles = $res['articles'];
```

4.12. 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->getMailbox();
$res = bw_exec( $player, "webGetPlayerInfo" );
```

For details on how this is implemented in BigWorld Python script, see “Retrieving real-time player data” on page 9 .

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

4.13. 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`:

```
$mailbox =& $this->auth->getMailbox();
...
$this->inventory = bw_exec( $mailbox, "webGetInventoryAndGold" );
```

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 = bw_exec( $mailbox, 'webCreateAuction',
```

```
    $itemSerialToAuction, $expiry, $bidPrice, $buyout );
```

4.14. 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 = bw_exec( $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.

4.15. 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 = bw_exec( $this->auctionHouse,
    "webCreateItemTypeCriteria", $itemTypesList );
...
$searchCriteria = $res['criteria'];
...
$res = bw_exec( $this->auctionHouse,
    "webCreateBidRangeCriteria",
    $searchMinBid, $searchMaxBid );
$bidRangeCriteria = $res['criteria'];
$res = bw_exec( $this->auctionHouse,
    "webCombineAnd",
    $searchCriteria, $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 = bw_exec( $this->auctionHouse, 'webSearchAuctions', $searchCriteria );
...
$res = bw_exec( $this->auctionHouse, 'webGetAuctionInfo',
    $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;
}
...
```