developerWorks®

# Building a simple yet powerful MMO game architecture, Part 2: Gaming and Web integration

Hyun Sung Chu

October 21, 2008

Massive multiplayer online (MMO) virtual-world games offer tantalizing new ways to learn, entertain, collaborate, socialize, visualize information, and do business. In this series, learn about an architecture based upon the first 3D MMO game from IBM®, PowerUp. Integrating a Web back end with a multiplayer online game (MOG) is a straightforward, effective way to provide MMO functions, such as persistence and integration. This article explores technical details of the architecture, including the functions, and calls for integrating game clients and servers with back-end systems.

View more content in this series

## Introduction

Part 1 of this series introduced a flexible and powerful MMO game architecture, based upon the fictitious MMO game Starship IBM, that's quick and easy to implement. It explained the implementation of the architecture and the interactions among gaming clients, gaming servers, Web servers, and a database.

This article explores the relationship between the gaming components and the back-end component and demonstrates how you can achieve a simple yet powerful MMO game architecture.

## Web back-end use cases

Most multiplayer gaming engines, such as Torque Game Engine Advanced (TGEA), provide default MOG functions out of the box. A Web and database back end can be used to add MMO functions. Table 1 shows the core subset of use cases provided by the Starship IBM Web back end. This subset can be used to implement any MMO game Web back end.

### Table 1. Starship IBM core Web back-end use cases

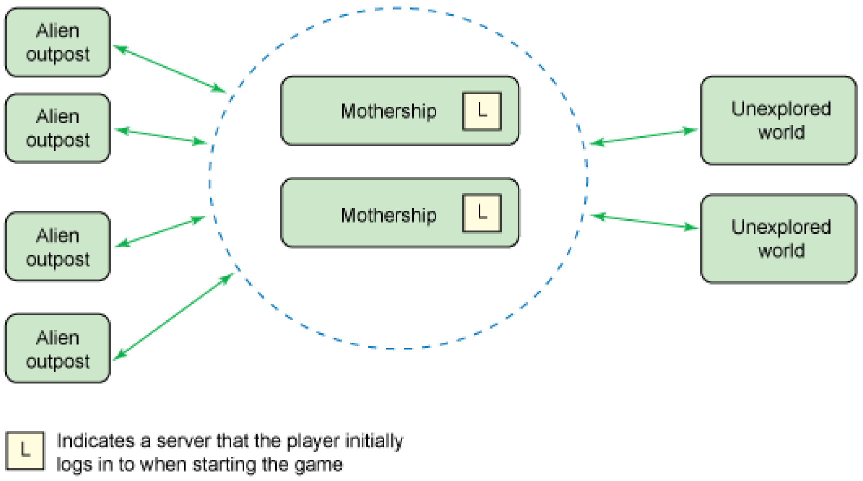
| Use case | Actor | Description |
|---|---|---|
| **Get available mothership server** | Gaming client | Retrieve the IP address and port of an available mothership server to initially log |

| | | |
|---|---|---|
| | | in to. This information is then used by the "Log in to server" use case to log in to the specified server. Additional game logic can be performed, such as gaming server load balancing, that determines which mothership server information to return. |
| **Log in to server** | Gaming server | Given the player's username, password, and the game server IP address and port, authenticate the user information. If authenticated, the database will be updated to indicate the player is logged in to this particular game server. |
| **Get a list destination servers** | Gaming client | Return a list of alien outpost and unexplored planet server IP addresses and ports and the number of players currently on each game server. The destination IP address and port can then be used in conjunction with "Check if server is full" and "Log in to server" to allow the player to travel from the mothership server to another game server.<br>The server player number can be used by players to load balance themselves (players can choose a game server to join that is not maxed out). |
| **Check if server is full** | Gaming client | Given the server IP address, this use case is intended as a last-minute check by the gaming client before initiating the loading and login to another server. |
| **Remove player from server** | Gaming server | If the player disconnects from the game for any reason, update the database to indicate the player is no longer on the game server. |
| **Get player points** | Gaming client | Given a username, return that player's point total. |
| **Update player points** | Gaming server | Given a username and point value, add that point value to the player's point total. |

*Actor* denotes whether the gaming client or server invokes the HTTP request to the Web server, not whether the gaming client or server initially invokes the use case. For example, the "Remove player from server" use case's actor is the gaming server. Even though this is initially invoked when, for example, the player chooses to quit the game from the gaming client, the gaming server should actually make the HTTP request to the back-end Web server.

Figure 1 shows the flow of the players in Starship IBM, and how they can travel from one server to another. All 20 alien outpost servers are not shown.

## Figure 1. Starship IBM player server flow



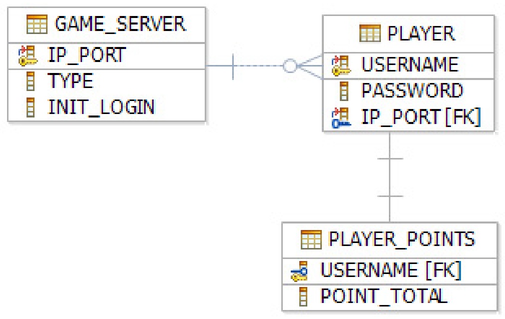
To help illustrate the flow, the following is a typical sequence of how a player goes through the various servers.

1. The player enters a username and password, and selects **Login** to start the game. This will initiate the "Get available mothership server" use case, which will retrieve the IP address and port of an available mothership server.
   Using this IP address and port, "Log in to server" will be invoked, which will authenticate the player and update the database to indicate the player is currently on the specified mothership server.
2. The player will bring up an interface that retrieves a list of available alien outpost servers, which will invoke the "Get a list of destination servers" use case.
3. The player selects one of the alien outpost servers and starts that mission, which will perform "Check if server is full" and then "Log in to server" using the IP address and port from the selected alien outpost server.
4. After the player is finished with the alien outpost mission, the player can go back to the mothership, which will perform essentially the same action as item 1 in this list. This time, though, the player will not have to enter a username and password.

## Database implementation

Figure 2 shows all the database tables that underlie the Web back-end use cases described in Table 1.

## Figure 2. Database implementation



The GAME_SERVER table stores the data for all the TGEA game servers. Note the one-to-many relation between the GAME_SERVER entries and the PLAYER entries. Each gaming server hosts zero to multiple players. There is also a one-to-one relation between the PLAYER table and PLAYER_POINTS table, as each player has a point total.

## Table 2. GAME_SERVER columns

| Column | Description |
|---|---|
| `IP_PORT` | The primary key (PK), consisting of the <IP address>:<Port>, which uniquely identifies each of the TGEA servers. An example value is `'0.0.0.0:28000'`. |
| `TYPE` | Indicates whether the server is a mothership, alien outpost, or unexplored world server. The allowed values are `'MSHIP'`, `'ALIEN'`, and `'UNEXPLOR'`. |
| `INIT_LOGIN` | Indicates whether this is a type of game server that the player can initially log in to (is it a mothership server). Allowed values are `'Y'` or `'N'`. |

The PLAYER table stores the base player information.

## Table 3. PLAYER columns

| Column | Description |
|---|---|
| `USERNAME` | The PK that uniquely identifies each player. An example value is `'Space Cowboy'`. |
| `PASSWORD` | The player's password. |
| `IP_PORT` | The foreign key (FK) to the GAME_SERVER table that indicates which gaming server the player is currently logged in to. |

The PLAYER_POINTS table stores point related data for each player.

## Table 4. PLAYER_POINTS columns

| Column | Description |
|---|---|
| `USERNAME` | The FK to the PLAYER table. |
| `POINT_TOTAL` | The player's point total. |

Given the GAME_SERVER table described above, and the actual Starship IBM server implementation described in Part 1 of this series, Table 5 below shows what the actual GAME_SERVER table entries would be. Of course, the IP addresses and ports are only examples.

### Table 5. Starship IBM GAME_SERVER table entries

| IP_PORT | TYPE | INIT_LOGIN |
|---|---|---|
| `'0.0.0.0:28000'` | `'MSHIP'` | `'Y'` |
| `'0.0.0.1:28000'` | `'MSHIP'` | `'Y'` |
| `'0.0.0.2:28000'` | `'ALIEN'` | `'N'` |
| `'0.0.0.2:28001'` | `'ALIEN'` | `'N'` |
| `'0.0.0.3:28000'` | `'ALIEN'` | `'N'` |
| `'0.0.0.3:28001'` | `'ALIEN'` | `'N'` |
| *Alien outpost servers 0.0.0.2:28002 - 28009 and 0.0.0.3:28002 - 28009 not listed* | | |
| `'0.0.0.4:28000'` | `'UNEXPLOR'` | `'N'` |
| `'0.0.0.5:28000'` | `'UNEXPLOR'` | `'N'` |

## MMO game Web application programming interface (API)

The Web application server uses the given database tables and serves up a set of MMO game functions. The API of the functions consists of HTTP GET requests to the Web server, and the Web application server returns an HTML page for a given request. The syntax of the HTTP GET request API is as follows:

```
[scheme]://[Web server IP address]/[path]?[query]
```

`[scheme]://[Web server IP address]/[path]?` is essentially static. The interesting part of the syntax is the `[query]`, which consists of `[action parameter][parameters]`.

`[action parameter]` consists of `action=[action name]`
`[parameters]` consists of 0 through N `&[parameter key]=[parameter value]` pairs.

This basic syntax can be used to accommodate all the HTTP GET request APIs. Table 6 shows examples of HTTP GET requests. Considering the Starship IBM architecture implementation, lets assume the public IBM WebSphere® server IP address and port are 1.1.1.1:80, and the secure WebSphere server IP address and port are 1.1.1.2:80.

## Table 6. Example HTTP GET requests

| Web back-end use case | HTTP GET request |
|---|---|
| **Get available mothership server** | `http://1.1.1.1:80/starshipIBM?action=getMothershipServerIPPort` |
| **Get player points** | `http://1.1.1.1:80/starshipIBM?action=getPlayerPoints&userName=Space Cowboy` |
| **Update player points** | `http://1.1.1.2:80/starshipIBM?action=addPlayerPoints&userName=Space Cowboy&points=200` |

Given an HTTP request, the Web application server will return an HTML page that the TGEA gaming client or server processes. The syntax of the returned HTML page can be:

```
request-status=[request status]
[key value pairs]
```

`[request status]` indicates the status of the HTTP request. The TGEA client or server will use this return status message and process the HTML page appropriately. For example, `SUCCESS` indicates the request was processed successfully, while other possible return status messages, such as `SERVER_FULL`, can indicate other states or problems with the HTTP request.

`[key value pairs]` consists of 0 or more `[key]=[values]` pairs. Each `[key]=[values]` pair is on a separate line. `[values]` can consists of one or more entries delimited by a comma.

Table 7 lists the returned HTML pages for the example use cases.

## Table 7. HTML page returns

| Web back-end use case | HTTP GET request query | Example returned HTML page | Notes |
|---|---|---|---|
| **Get available mothership server** | `action=getMothershipServerIPPort` | `return-status=SUCCESS`<br>`ip_port=0.0.0.0:28000` | |
| **Log in to server** | `action=loginToServer`<br>`&destServerIP=0.0.0.0:28000`<br>`&username=Space Cowboy`<br>`&pw=43YaC0nE9c` | `return-status=SUCCESS` | The Web server doesn't perform the actual game login and loading. It simply authenticates the player and updates the database to indicate which server the player is on. |
| **Get a list of destination servers** | `action=getServerList` | `return-status=SUCCESS`<br>`server_info=0.0.0.2:28000,ALIEN,8`<br>`server_info=0.0.0.2:28001,ALIEN,5`<br>`server_info=0.0.0.3:28000,ALIEN,0`<br>`server_info=0.0.0.3:28001,ALIEN,2`<br>`server_info=0.0.0.4:28000,UNEXPLOR,32`<br>`server_info=0.0.0.5:28000,UNEXPLOR,35` | The `server_info` values are the gaming server IP address:port, server type, and number of players currently on the server. `ALIEN` servers 0.0.0.2:28002 - 28009 and 0.0.0.3:28002 - 28009 are not listed. |
| **Check if server is full** | `action=checkServer`<br>`&ipAddr=0.0.0.4:28000` | `return-status=SUCCESS`<br>or:<br>`return-status=SERVER_FULL` | |
| **Remove player from server** | `action=logout`<br>`&username=Space Cowboy` | `return-status=SUCCESS` | Should be invoked when a game client disconnects from a game server for any reason. |
| **Get player points** | `action=getPlayerPoints` | `return-status=SUCCESS` | |

| | | | |
|---|---|---|---|
| | `&userName=Space Cowboy` | `points=12400` | |
| **Update player points** | `action=addPlayerPoints`<br>`&userName=Space Cowboy`<br>`&points=200` | `return-status=SUCCESS` | |

The main characteristics of an MMO game are *persistence* and *integration* of the gaming world and servers. The use cases in this article demonstrate how to achieve persistence and integration in the Web back end. The "Get player points" and "Update player points" use cases show how to achieve persistence. It is easy to then extrapolate other persistence related functions, such as retrieving and updating a player's inventory or game world status.

Gaming world integration is more interesting and a little more involved. The small set of server-related use cases in this article show how gaming world integration functions can be provided by the Web back end.

### Example: Log in to server in Torque

Implementing the use cases is straightforward using TorqueScript. Create a TorqueScript `HttpObject`, make the appropriate `HttpObject get` request call, and process the returned HTML page. However, implementing the "Log in to server" use case might not be that obvious. To go from one server to another in Torque, you need to invoke the standard TorqueScript client functions `disconnect()`, then `GameConnection.connect(%ipAddr)`, where `%ipAddr` is the destination server IP address. This will disconnect the player from the current game server and start loading the new mission from the new server. The "Log in to server" Web request can then be invoked in the default TorqueScript server function `GameConnection::onConnect()` to perform its housekeeping duties of properly updating the database.

"Check if server is full" should be invoked right before `disconnect()`. The sole purpose of this use case is to help reduce the number of unnecessary (and unceremonious) dumping of game clients from the game to the login screen. This happens when `disconnect()` is called before `GameConnection.connect(%ipAddr)`. A typical, preventable case occurs if the destination server player count maxed out before the client could start connecting to that server.

## Other Web back-end use cases

The small set of core back-end use cases in this article helped explain the MMO game function provided by the Starship IBM Web back end. Of course, an actual MMO game would include more involved functions. The following list has brief descriptions of other possible MMO game use cases that can readily be implemented with the Web back end described in this article.

**More persistence functions**
Besides retrieving and updating points, other persistence functions can also be implemented, such as retrieving and updating player profiles, inventory, and game world status.

**Viewing a list of players on other servers**
Since the player and server relationship is tracked in the database, it's straightforward to display the list of all the players on a particular server. Messaging players on other servers is also an intriguing possible function; Part 3 of this series will explain how it might be implemented.

**Mission-status related functions**

The game servers can schedule a recurring function that updates the Web and database back end with its mission time.
For example, the alien outpost mission could be a timed mission. The mission time can be passed to the Web and database back end at a repeated interval.

**Game server monitoring**

All game servers can schedule a repeated keep-alive message to the Web and database back end. In conjunction with an e-mail messaging component in the Web application server, this message can be used to notify system administrators when a game server goes down. The server status information can also be incorporated into any game server load-balancing algorithm, so users are not sent to a game server that is apparently down.

**Integration with other applications**

There are intriguing possibilities for further integration with other applications, using a middleware application such as IBM WebSphere.

## Summary

A Web and database back end can be used to provide MMO game functions, such as persistence and an integrated game world. You can do so in a straightforward manner using a set of Web APIs using HTTP GET requests. These APIs are part of a standard three-tiered architecture, where the gaming client and servers cleanly integrate with a Web and database back end. The APIs have an effective array of options that can provide virtually all MMO game-related functions.

Stay tuned for Part 3, which will review and assess the capabilities and limitations of the MMO game architecture, and whether it meets the high-level specifications and goals defined in Part 1 (scalability, flexibility in deployment, flexibility in gaming design, performance, and security).

## Acknowledgments

Thanks to Scott Crowther and Abraham Guerra from IBM CIO for implementing the WebSphere and DB2® components for PowerUp the game. This allowed the author to focus on such business critical tasks as dune buggy anti-rollover logic, water balloons, laser beams, and "smog monsters."

## Related topics

- Check out the other parts of this series:
    - Part 1 shows you how to build a flexible and powerful MMO game architecture that is quick and easy to implement.
    - Part 3demonstrates how the MMO game architecture meets the high-level architectural goals of scalability, flexibility in deployment, flexibility in gaming design, performance, functions, and security.
- Read about MMOs on Wikipedia.
- GarageGames specializes in gaming engines targeted for small and independent game developers.
- Torque Game Engine Advanced (TGEA) is GarageGame's flagship 3D game engine.
- Torque Game Engine (TGE), another game engine from GarageGames, has less advanced graphical capabilities than TGEA. It cannot render shaders (out of the box), but its networking capabilities are very similar to TGEA. The architecture described in this article does not use any particular feature that is TGEA specific and not part of TGE. TGE also runs on Linux® and Macintosh.
- The GarageGames forum covers Torque client/server networking.
- Read about how Torque interacts with a Web server.
- For comparison purposes, Unity is a 3D gaming engine that has features comparable to Torque.
- Check out Unity's game networking, particularly the section on "WWW functions" to learn how Unity talks with Web servers.
- Learn more about the various IBM products described in this article, including IBM WebSphere , IBM DB2 , and IBM BladeCenter® servers.
- Get the RSS feed for this series.
- Download, explore, and play PowerUp.