Database Driven Networking Kit Documentation

By: Ignition Games

Version 1.0

Table of Contents

Introduction

Welcome to the Database Driven Networking Kit for Unity! In this kit you'll find many easy to follow examples on how to connect Unity to an external PHP/Database server to drive your game. Many games made today use this architecture, for example Clash of Clans, Game of War and Boom Beach just to name a few. With this kit you will be able to quickly and easily make games like these by having all of the server communication code already done for you.

It is assumed that you the developer have an intermediate level of knowledge of the Unity engine and some experience with PHP and mySQL databases. All example code in this documentation and kit are written in JavaScript.

The Database Driven Networking Kit (DDNK as we will refer to from now on) contains examples for Mobile and PC but can be used with other platforms that accept Unity's built in WWW class. You will have the ability to load data from any external web server running PHP and store it on a GameObject for later reference and use.

* Unity's Social API currently does not support Google Play. A separate plug-in is required.

Included examples:

- Simple Connection
  - Shows the basics for connecting a Win/Mac/Linux game. Displays the ability to load a username, example string and last saved position. When the data is loaded in Unity, an example function is called to change the position of a box to the position loaded from the database. The assigned player ID in the database is saved using Unity's PlayerPrefs.
- Simple Mobile Connection
  - Functions the same as the Simple Connection, but instead of using Unity's PlayerPrefs to save the ID of the player, the GameCenter/Google Play* user id is loaded and used so no player registration is required. Like the above example, a position is loaded from the database and applied to a box in the level.
- World Loader (With EventQueueManager)
  - This example is useful for anyone looking to create a game that requires dynamic loading of things such as buildings and other player specific variables. This example is for anyone who needs loading and saving capabilities like Clash of Clans.

    The World Loader example displays a loading screen while the players' data is being loaded from the server and then hides it when this process is complete. Building positions are loaded and then placed while the loading screen is being shown, and other variables such as gold and diamond amounts are loaded as well. In conjunction with the EventQueueManager, you can also build new buildings and have them save the database to be loaded each time the game is run.
- Event Queue Manager
  - The Event Queue Manager is a class that makes it easy to send new "event" updates back to the PHP server and database for processing and storage. Our

example links in with the World Loader to allow a player to click a button to place a building at a random position in front of the camera, and have the building name and position sent back to the PHP scripts to be saved in the database. When loading the mission each time after, all of the buildings created will be dynamically loaded and placed while the mission loads.

This class can be easily built onto to allow for any type of event update that you want to save to the database.

Unity to PHP to Database and back to Unity

DDNK uses Unity's WWW class to talk with a series of PHP files that can send and receive any type of data to and from Unity. Our examples show how you can use PHP files to load data from a database and send it back to Unity. The PHP files serve a bigger purpose than just a relay between a database and Unity; they also allow a developer to do data checking to make sure the data being sent from Unity and stored in the database is in fact real and not manipulated.

The way this all works is a class is first setup to store the received data in, well make an example below:

```
class dataStorage
{
        var testString:String = "";
        var testNumber:int = 0;
}
var ourDataObject:dataStorage = new dataStorage();
```

So now we can access the instance of this class in our script simply by using ourDataObject. What DDNK does is loads some pre-defined data from a database and then stores it in a class' variables like the one above. It's completely up to the developer what kind of data they want to load and store but in this example here, we will be making a game like Clash of Clans. We want to know what kind of buildings we have and where they are positioned and we want to know how much gold we have. The World Loader example is built around this idea so use it as a reference when following along.

First we setup a storage class with all of the data that we are going to want to receive. The variables can be in any order, it doesn't matter. When the level is loaded in Unity, the `Start()` function is called which calls either `connectLogin()` or `connectSocial()` depending on which example you are using. Those functions both do the same thing, get an ID that the player has been assigned so that when we want to load our data, the PHP code knows which set of data were sending back or updating based on the players ID.

For security purposes, we store a "private key" string in the PHP code and in each connection script. Unity sends this stored private key in the WWW requests and it is a variable at the top of each script called `UNITY_PRIVATE_KEY` and it must match the key defined in Server Backend/PHP/includes/server_settings.php called `$GAME_PRIVATE_KEY`. If the keys don't match, a "criticalerror – key mismatch" will be thrown.

Next after we know the ID were sending that is associated with our player's database row, we call download_data() which uses Unity's WWW class to send off a request to our PHP file for the information we want. After the WWW request is successful, we send the received response which looks like this:

server_time=1430857914;data_social=1000;data_username=Lerpz;data_exampleDatabaseVariable=Some String;data_lastPosition=(0,5,0);

to a function called split_data_into_array() which splits the above returned string of all our data into segments so each piece of data can be read. It is VERY IMPORTANT that the order of your database columns EXACTLY match the ordering of checked variables inside the split_data_into_array() function. This is a security feature as well as a way to make sure that the data sent from the PHP files to Unity is correct and exactly what we want. Additionally, the names of your database table rows are important as well. The way the PHP function returnPlayerData() in Server Backend/PHP/includes/include-functions.php is setup, it goes through all table columns and checks their name. If a table column contains "data_", it is sent to Unity. This is setup so that important data that we don't want/need to send to the player is secure and can't be viewed.

split_data_into_array() first splits the string by semicolons(;) which will essentially be each variable we want, then splits each line by equals sign (=) which will result in getting the values for each variable we want to store. For example, the above string is a returned string from our Simple Mobile example. It includes the username (…data_username=Lerpz;…) of our GameCenter username "Lerpz". Referring back to the class we created above, you could add a variable to that class called username, and we could save this received username to the class for further reference at any time.

DDNK also includes a class called eventQueueManager that allows you to send data to the eventQueue.php file to be saved in a database. The process for this is you send your data to a function called addUpdateQueueEvent() with parameters in a specific order. There are 5 parameters this function could take, and all of them are strings. The first one should always b the "event type" you are sending. In our eventQueueManager script, we check for an event type of "createBuilding" with the second argument being the building name (just the name of the prefab for that building) and the third argument being the position of the building in string form. This function then calls updateEventQueue() which sends data to eventQueue.php, gets checked and then updated in a database. Any time an update is sent to this PHP file, it returns back the same response as above in blue, and split_data_into_array() is called on the response again to update the new data that has been received. This is useful so a player can't cheat, and any data issues that PHP finds when receiving an update gets corrected, saved and resent back so the players game state is always in sync with the information in the database.

To recap on the above; when our level is loaded, DDNK is activated and it sends an initial connect request to a PHP file that returns back a string of data from a database that contains the variables and values we want to store. The string is split and each variable gets extracted and read in an expected order, then stored in a class for further use or reference later.

Server Backend setup and requirements

The heart of DDNK is the external PHP and database server(s) and before Unity can load any data, we need to upload our PHP files somewhere and setup our database. If you don't know much about PHP or databases for the matter, it is highly recommended that you read up on the subjects to get a better understanding of how they work. For this documentation, enough instruction will be given to explain how to setup the server backend for our use.

First, you need to upload all of the PHP files located in the Server Backend/PHP folder, to a PHP enabled web server. If you have a website somewhere, then you more than likely have everything you need. If you don't, you can use the default Ignition Games net services url included in all of the script links. You won't have the ability to make any changes to the PHP code or the database if you use the Ignition Games urls. If you have your own website, simply create a new folder or even a new sub-domain and upload everything contained in Server Backend/PHP.

Next you will need to setup your database. This documentation is strictly for explaining how DDNK works and how to use its features, so if you don't know how to create a database on your web host, do a Google search and you will find millions of articles that will give you a step by step guide.

When setting up your database, you will need to note the hostname (usually localhost if on the same server as your website files), the database name, username for your database and password. You will need to set these variables in the server_settings.php file located in Server Backend/PHP/includes/

Once your database is setup, you will need to create your table(s). Again, if you don't know how to do this, simply search Google for "Import sql file into database". More than likely your web host will have an article on how to import sql files into a database. If you have the ability to use phpMyAdmin, just select your database, click the SQL tab at the top and paste the contents of the .sql file that is required for the example you wish to run.

Once your database and table(s) are setup, you just need to change the "url_string" variable in the script file or from the Inspector in the Unity Editor, to your url to the correct PHP file on your own server.

If you do not have a web server that can be used and wish to rent one, AmbientWave has partnered with Ignition Games to offer Unity Game Developers web hosting for using DDNK for only $5/mo. Just visit their contact page and mention that you are a Unity Developer using DDNK and are in need of hosting.

Examples

Simple Connection

Files for this example:

- SimpleConnectionTest.unity
  - The main scene file for this example
- Connection Examples/SimpleConnection.prefab
  - This contains the simple connection script that loads some basic variables

- Server Backend/PHP/manager.php
    - This is the PHP file the simple connection script communicates with
- Server Backend/SQL/accounts.sql
    - This is the SQL import file for the database table that is compatible with manger.php

Recommended Platforms: Standalone (Win, Mac, Linux), Unity WebPlayer

In this example we demonstrate how to create a simple connection between Unity and a database to load variables that we can use to display or apply to objects in the game world. This example moves a cube to a position that is loaded from a database, and loads some other basic variables for demonstration purposes.

To use this example, follow the steps in the **Server Backend setup and requirements** section first. Next you can drag the SimpleConnection prefab in the Connection Examples folder, into your scene. All you need to do now is push the Run button in the Unity Editor and your data should be loaded from the database, and a box named "moverObject" is moved to the position of the "data_lastPosition" return variable.

This example is very primitive in how the player is assigned an ID for the database. Unity sends a PlayerPref variable called "userID" which if not sent is sent as 0. The PHP file looks for a database row with the user_id of 0, and can't find anything. Like all of the PHP manager files, if a database row with the id being searched for isn't found, a new row is created and that ID and data is returned back to Unity. In our example, we set a PlayerPref called "userID" in the same place we set the id variable of the simpleConnection class, so next time the game is loaded, the userID PlayerPref will be read and the correct database row is returned and loaded in Unity.

Simple Mobile Connection

Files for this example:

- SimpleConnectionMobile.unity
    - The main scene file for this example
- Connection Examples/SimpleMobileConnection.prefab
    - This contains the simple connection script that loads some basic variables
- Server Backend/PHP/manager_mobile.php
    - This is the PHP file the simple connection script communicates with
- Server Backend/SQL/accountsMobile.sql
    - This is the SQL import file for the database table that is compatible with manger_mobile.php

Recommended Platforms: iOS, Android, Windows Phone, Blackberry

In this example we demonstrate how to create a simple mobile connection between Unity and a database to load variables that we use to display or apply to objects in the game world. This example moves a cube to a position that is loaded from a database, and loads some other basic variables for demonstration purposes.

To use this example, follow the steps in the **Server Backend setup and requirements** section first. Next you can drag the SimpleMobileConnection prefab in the Connection Examples folder, into your scene. All you need to do now is push the Run button in the Unity Editor and your data should be loaded from the database, and a box named "moverObject" is moved to the position of the "data_lastPosition" return variable.

This example uses Unity's Social Platform API to connect to and get the user id from that specific platform. *Currently Unity's Social API only works out of the box with Apple's GameCenter so Andriod developers needing to connect to Google Play will need a separate plug-in that allows access to Google Play Game Services.* Personally I use this, it's free and works perfectly. Once Unity successfully connects to the Social Platform and receives the user id, it creates a connect request with our Social Platform ID to manager_mobile.php where we check the database rows for that ID. Like all of the PHP manager files, if a database row with the ID being searched for isn't found, a new row is created and that ID and data is returned back to Unity. In the mobile example, the Social Platform connection is always made before we connect to our database so the same ID is being used every time. This allows you to make mobile games that require no profile registration and are secure, because GameCenter/Google Play tells Unity what the players ID is so each player is unique.

World Loader & Event Queue Manager

Files for this example:

- WorldLoader_with_EventQueue_Buildings.unity
    - The main scene file for this example
- Connection Examples/WorldLoader.prefab
    - This contains the world loader connection script that loads buildings and variables
- Connection Examples/EventQueue.prefab
    - This contains the script that handles all updates being sent to the PHP file/database
- Server Backend/PHP/manager_worldLoader.php
    - This is the PHP file the world loader connection script communicates with
- Server Backend/PHP/eventQueue.php
    - The PHP file in charge of receiving event updates from Unity
- Server Backend/SQL/accountsWorldLoader.sql
    - This is the SQL import file for the database table that is compatible with manger_worldLoader.php
- Example Objects/LoadingScreen.prefab
    - The prefab for a simple loading screen canvas
- Example Objects/buildingManager.prefab
    - The prefab with the building manager script
- Example Objects/buildingCanvas.prefab
    - The prefab that contains a canvas with a button for adding buildings

Recommended Platforms: Standalone (Win, Mac, Linux), Unity WebPlayer, iOS, Android, Windows Phone, Blackberry

In this example we demonstrate how to instantiate buildings from loaded data and move them to their loaded positions. It also loads variables that represent gold and diamond amounts. Additionally it includes a button that creates a building in a random position in front of the camera, and then sends updates with the new buildings position to be saved and loaded from the database for that player. In this example, 1 gold and 1 diamond get subtracted from the inventory each time a building is created. If no gold/diamonds remain, a building is not created.

To use this example, follow the steps in the **Server Backend setup and requirements** section first. Next load the WorldLoader_with_EventQueue_Buildings.unity scene an push Run in the Unity Editor. Unity will connect to the PHP file and load the data for the player, and then instantiate buildings in the level from the loaded data. A button in the lower left corner of the screen will appear that will create and save the new building and its position to the database. Each subsequent run will load all of the newly created buildings.

This example uses the same user id storage as the Simple Connection Example with Unity's PlayerPref class so you will want something more robust and secure for a final product.

Building off of an example to load your own data

This section will go over step-by-step how to use the Simple Connection Example to load new data and do something with it. We will also go over the eventQueueManager to add new events. This will include changes to the Unity script, as well as what to do with the database and PHP file.

In this example, let's say we want to make a simple game of chess. This game will be single player, but we want to be able to keep track of wins, losses, least number of moves and fastest round win time. Our example will assume that the actual chess game is complete and we are able to tell if we have won or lost, and also a variable has been created to use as a reference to how long the round lasted.

First, let's start with the database side. You will need to know how to change a database table's structure so a Google search might be required. Like was mentioned earlier in **Server Backend setup and requirements** the name of your database table columns matter. To keep with our example, create new columns called "data_wins", "data_losses", "data_leastmoves" and "data_lowestroundtime". The "data_" is used by the PHP file to send back our variables to be loaded and saved in Unity. A column needs to have "data_" in its name in order for Unity to receive this variable. You can see how this works by looking at the resendPlayerData() function in Server Backend/PHP/includes/include-functions.php. Also note the order in which you create these new columns as this will matter later when we change the Unity script on our variable parse order.

Our PHP code in manager.php doesn't need to be modified as it will fetch our new columns from the database and return them to Unity. If the database changes are done correctly as mentioned above, the PHP output will look like so:

```
server_time=1430871528;data_social=1000;data_username=Lerpz;data_exampleDatabaseVariab
le=Some
String;data_lastPosition=(0,5,0);data_wins=0;data_losses=0;data_leastmoves=0;data_lowestroundt
ime=0;
```

Next we need to modify our Simple_Connection.js file located in Scripts. Because we modified our database table, we need to modify the `split_data_into_array()` function to take into account the new variables we are loading as well as add some new variables to our `c_connObject` class.

First let's add the new class variables. Assuming only additions to table were made; we would add new variables like such:

```
…
var wins:int = 0;
var losses:int = 0;
var leastMoves:int = 0;
var lowestRoundTime:float = 0.0;
…
}
```

Next inside the `split_data_into_array()` function's switch statement, we need to add new cases for each new table column. Assuming only the above additions were made to the table, and no columns were removed, this is the new example code to add to the switch statement:

```
case 5: // c_connObject.wins
        serverConnection.wins = int.Parse(fields[1]);
case 6: // c_connObject.losses
        serverConnection.losses = int.Parse(fields[1]);

case 7: // c_connObject.leastMoves
        serverConnection.leastMoves = int.Parse(fields[1]);

case 8: // c_connObject.lowestRoundTime
        serverConnection. lowestRoundTime = float.Parse(fields[1]);
```

fields[] is created by splitting each variable by the equals sign (=), with `fields[0]` being the "data_wins" for example and `fields[1]` being the value in the database. You must remember the data type of each variable in the storage class as if assigned a value, must match that data type it's being assigned to. Just remember that `fields[1]` is always a string, and if a variable like `serverConnection.wins` is an integer, you need to use `int.Parse()` or else an error will be thrown and no further data is loaded.

At this point, we should now be able to load our new data, and you can click on the SimpleConnection gameObject to view the loaded variables. All our variables will be 0, and will stay that way unless we actually send event updates to our server. This is where the eventQueueManager comes into play. Drag the EventQueue.prefab file (located in Connection Examples) into your scene. By default the eventQueueManager is setup to work with the World Loader example so we will need to make a few changes.

First on line 3 of the eventQueueManager.js file (located in Scripts) we will need to modify which connection object we are trying to find. It should be looking for the "WorldLoader" gameObject, change this to "SimpleConnection". This link allows us to do 2 things, we need the user id that is linked to our database table row for our player, and we need to be able to call split_data_into_array() on our newly updated data we will receive when sending an event update.

Next go down to the addUpdateQueueEvent() function. Our first argument in the function called "type" is used to tell what kind of event we are adding. Notice the if statement checking to see if the type is "createBuilding". In the World Loader example we use this to format our event update into a format to send the event type, building name and position "createBuilding building1 (x,y,z)". This string of space separated words is read by the PHP code to also tell what type of update its dealing with, and then add the additional information to the building column.

So for our example, remove the if statement for "createBuilding" and we will add new code in its place that will look like so:

```
if(type == "addWin")
{
	// arg1 = number of moves
	// arg2 = time of round
	updateQueueObject.queue.Add(type+" "+arg1+" "+arg2);
}
else if(type == "addLoss")
{
	updateQueueObject.queue.Add(type);
}
```

Next from the game logic, we need to reference this script so the following code will need to be added to the top of the script along with other variables:

```
var eventQueue:eventQueueManager;
eventQueue =GameObject.Find("EventQueue").GetComponent(eventQueueManager);
```

This will allow us to call the addUpdateQueueEvent() function by using the eventQueue variable like so:

```
eventQueue. addUpdateQueueEvent();
```

Again still our game logic; in some section of code that gets called when a player wins we would add this:

```
eventQueue. addUpdateQueueEvent("addWin", numMoves.ToString(), roundTime.ToString(), "", "");
```

The numMoves and roundTime will need to be created and set by you. Also remember that all arguments in addUpdateQueueEvent() are strings so note that all non-string parameters need to be converted into a string using ToString(). Also there are no overloads for this function so all 5 string parameters are required to call it. Note the empty strings ("") sent as the other blank parameters.

Still in our game logic, we need to add the following line to code that gets called when the player loses:

```
eventQueue. addUpdateQueueEvent("addLoss", "", "", "", "");
```

The reasons were not sending any numeric parameters in our event updates is that this removes a cheating situation. We will let the PHP code increase our wins or loses depending on which event is received. So how do we tell when numMoves and/or roundTime is our lowest and if we should update the database to these values? Well again we let the PHP handle this for the same reasons as previously mentioned. On to the PHP code!

The eventQueueManager script talks to the eventQueue.php file (located in Server Backend/PHP). We will need to modify a few things here. First we need to make sure that we are using the correct table that we modified for our changes. Anywhere where "accountsWorldLoader" is mentioned, we need to change this to "accounts". If you modified a different database table, change the name to the correct one. There are 2 instances of this in eventQueue.php. Next in the foreach loop, you might notice something familiar:

```
if($r[0] == "createBuilding")
```

This works the exact same way as the Unity script we modified earlier, only this is the receiving end. The $r[] variable is an array of the space separated string that was created in the previous script. For example, this PHP script receives a win update like so:

```
"addWin 125 500.34" (event type, number of moves, round time)
```

$r[0] would be "addWin", $r[1] would be "125" and $r[2] would be "500.34". You can see how this works in the current code. But before we modify this, we need to create variables for what our current wins, losses, number of moves and quickest time are. Youll notice above that we have a variable like so:

```
$data_buildings = $row["data_buildings"];
```

The variable is setup just like its matching database column. $row is an array variable of all columns and their values for our players table row. We want to remove this line and create our own:

```
$data_wins = $row["data_wins"];
$data_losses = $row["data_losses"];
$data_leastMoves = $row["data_leastmoves"];
$data_lowestRoundTime = $row["data_lowestroundtime"];
```

Now that we have easy access to what our current numbers are, let's go back down to that if statement and remove it. Just like in the Unity script, we need to add an if statement for each event type like so:

```
if($r[0] == "addWin")
{
        $data_wins++; // increase our wins
        // if the number of moves is lower than our current, update to new number
        if($r[1] < $data_leastMoves || $data_leastMoves == "")
                $data_leastMoves = $r[1];
        // if the lowest round time is lower than whats in the database, update to new time
        if($r[2] < $data_lowestRoundTime || $data_lowestRoundTime == 0)
                $data_lowestRoundTime = $r[2];
}
else if($r[0] == “addLoss”)
{
        $data_losses++; // increase our losses
}
```

The above code is pretty self explanatory, but we check what event type were dealing with using `$r[0]` then in our “addWin” if statement we check to see if our number of moves and round time are lower than our current ones OR if our current values are the defaults and this is the first played game. For “addLoss” we simply increment our losses.

Next and final step is to actually send the update command to the database. Lower down you’ll notice the following code:

```
// Update account info
$upd["data_buildings"] = $data_buildings;

$upd[] is an array that we use lower down in our sql update call:

$sql->update("accountsWorldLoader", $upd, "WHERE `user_id…...
```

So in order for updated variables to be sent and stored, we need to add some variables to this array. Note that the variable name in our `$upd[]` variable needs to exactly match the database table column were updating and the value is the variable we created an update above. Here is the new code below:

```
$upd["data_wins"] = $data_wins;
$upd["data_losses"] = $data_losses;
$upd["data_leastmoves"] = $data_leastMoves;
$upd["data_lowestroundtimes"] = $data_lowestRoundTimes;
```

After this, the `$sql->update()` function will be called and the table columns updated! The newly updated data will be sent back to Unity and updated in the storage class.

The previous example could be applied to any type of game and data. In a matter of 15 minutes, you can have all of your players variables being loaded at the start of the game and all updating functionality done. DDNK is an easy and fast way of getting your externally stored player profiles/stats/etc. loading and updating efficiently and securely.

Support

If you require support, have a bug to report or have any general comments, questions or suggestions you can visit http://www.ignition-games.com/contact.php and your inquiry will be answered promptly.

Thank you for purchasing DDNK!