Shared Wallet Interface Specification

Version 2.0.3

August 23, 2016

Contact support@rivalpowered.com

Table of Contents

RIVAL SHARED WALLET INTEGRATION

INTRODUCTION

The Rival Game Engine can operate standalone, or in cooperation with an external platform or “shared wallet”. This is also often referred to as a “Seamless Wallet Integration”. When using the game engine with a shared wallet, player wagers trigger authenticated requests to an external web service to efficiently share the player's wallet between the primary platform and the Rival Game Engine in real-time.

In order to integrate a shared wallet, Rival requires that a web service with the following API be provided for use by the Rival Game Engine.

API OVERVIEW

The Rival Game Engine handles all of the game logic, balance updates, error handling, and front end communication. Paired with the Rival Flash Front end, Rival Mobile client and powerful Back Office, this system is designed to be a complete solution for gaming.

The Engine API describes how the Rival system communicates with primary platforms for shared wallet integrations. Our system uses a simplistic model using only 5 calls to communicate actions with a remote web service. These calls are:

- **validate** : Used to authenticate a player
- **getbalance** : Used to retrieve a player's balance
- **updatebalance**: Used to debit or credit a player account
- **rollback**: Used to roll back a balance change operation in case of communications errors.

- **status**: Used to check if the service responds

ENVIRONMENTS

The Rival team expects that your team will work with two separate environments during the development and subsequently launch of your product:

- **Staging environment –** For testing and development.
- **Production Environment** – The live environment. No development happens here.

Our team will configure both environments for your purposes. We expect that you provide us with a separate configuration, including communication endpoints, for each environment.

COMMUNICATION

The Rival Game Engine can be configured to work with existing systems, but is set up to work nearly completely "out of the box" using HTTP POST requests/responses and JSON messages.

You must provide us with the hostname and path to your service that will accept requests from our Engine. This hostname and path will be used to form the complete URL to the service set up to respond to the Rival requests. An example:

> **Hostname**: `https://example.com`
> **Path**: `/serv/Rival`
> **Assembled URL**: `https://example.com/serv/Rival?jsoncall=validate`

Note: A separate endpoint URL must be provided for both *Staging* and *Production* environments.

As you may have noticed in the above example, we will append a `jsoncall` parameter to the URL which specifies the function being called. This `jsoncall` parameter will correspond to

one of the above mentioned calls.

After assembling the URL, the Rival engine will then make a POST request to your service, passing a single JSON-encoded object as a POST parameter. Within this object, you will typically find a list of parameters pertaining to the call. These parameters are the data that you will need in order to facilitate the integration.

We then expect your system to return a JSON-encoded response object containing data that will be used to coordinate actions within our software.

MESSAGE AUTHENTICATION AND SECURITY

It is recommended that messages sent between the Rival Game Engine and the primary platform be authenticated by passing a message digest alongside each message (e.g HMAC-SHA256).

This is a typical example, written in PHP, of how we would generate an *hmac* parameter and send it with a request:

```
$request = array();
$request[“playerid"] = 'testuser123';
$request[“sessionid"] = '6a204bd89f3c8348afd5c77c717a097a';
$request[“function"] = 'validate';
$httpQuery = urldecode(http_build_query($request));
$request['hmac'] = hash_hmac('sha256', $httpQuery, $secret);
```

All communications in a *Production* environment are expected to use secure HTTP (SSL/TLS). Ie: Your URL endpoint must begin with “https://” (*Staging* environment endpoints do not require this.)

LAUNCHING GAMES

Once the communication layer is complete, you may launch any enabled Rival game from the primary platform, passing in a valid authenticated account, and transactions will be communicated from the Rival system to the platform. Our suite includes both Flash-based and Mobile (HTML5) games. While the “behind-the-scenes” communication remains the same for either implementation, there are specific details that need to be followed in order to launch each type of game.

GAME IDS

Each game in our suite has a unique “root” game ID. That is the game ID that is used to launch the game. However, many of the games we offer have sub-games, bonus rounds, multiple game rounds, etc. For this reason, it is highly recommended that you implement the integration to allow for any game ID to be sent to your system. This also allows for easy integration of new games as they are added to our suite. For example: A game with root ID 3, may have 2 additional sub games with IDs 7 and 12 respectively. As a player progresses through our games, we will record transactions with the most accurate game ID possible.

FLASH GAMES

Flash games are browser-based games launched within a browser window. These games are natively sized to 800x600. This is the size that games should be launched in order to look their best.

Please obtain the correct launch URLs from Rival Support (support@rivalpowered.com). In general, the format for the URLs will look like this:

Staging:

http://demo.casinocontroller.com/beta/<casinoname>/engine/EmbedGame/EmbedGame.php?game_id=X&playerid=ABC&sessionid=123

Production:

https://www.casinocontroller.com/<casinoname>/engine/EmbedGame/EmbedGame.php?game_id=X&playerid=ABC&sessionid=123

Please note that for Flash Games, these URLs *must* be loaded in an iframe, otherwise the EmbedGame.php page will redirect itself to your main web site immediately. Also note that while we highly recommend loading your entire site over HTTPS, we realize this is not always possible. So, to avoid the mixed-content warnings in several web browsers, it is best to omit the protocol when specifying your iframe URL. That is, your URL should begin only with "//www.casino...." (see below)

You will be required to pass in game_id as well as playerid and sessionid. game_id is the Rival game identifier. For a full, current list of games and game_id's, visit https://admin.casinocontroller.com/marketing-materials . The parameters playerid and sessionid are parameters passed by your system and will be used in subsequent function calls, such as “validate”.

Here is an example iframe snippet:

```
<iframe height="600" width="800"
src="//www.casinocontroller.com/<casinoname>/engine/EmbedGame/EmbedGame.php
?game_id=42&playerid=4&sessionid=abc123"></iframe>
```

MOBILE GAMES

Launching mobile (HTML5) games differs only slightly from launching Flash games. Mobile games are designed to work on a variety of mobile devices, so there is no strict sizing requirements. We do recommend, however, that you add logic on the platform to recognize mobile users and only launch the mobile games for mobile devices, (preferring the Flash games for desktop clients).

It is important to note that while the Flash games must be launched within an iframe, mobile games **must not** be launched within an iframe. Our code has built-in size and orientation logic which is needed to display the mobile games correctly.

Please contact Support (support@rivalpowered.com) to obtain exact URLs for launching mobile games. In general, the URLs look like this:

Staging: http://demo.casinocontroller.com/beta/<casinoname>/html5/dist/?game_id=X&playerid=ABC&sessionid=123

Production: https://www.casinocontroller.com/<casinoname>/html5/dist/?game_id=X&playerid=ABC&sessionid=123

The parameters game_id, playerid and sessionid are the same as detailed above. Please note that only mobile game_id's can be used to launch mobile games and Flash game_id's to launch Flash games.

ADDITIONAL PARAMETERS

In the case of both Flash and Mobile games, you can pass additional GET parameters to our games to specify additional functionality, such as language, banner tags, tracking, etc. Here is a breakdown of the additional parameters available. These should be appended to the Flash launch URL and Mobile launch URL, as needed:

- **lang** – a 2-character language code specified by your system.
- **btag** – A banner tag generated by the Rival system. Can be used for tracking campaigns created in the Rival backend. Sent and recorded for new users only. (Arbitrary campaigns cannot be passed in)
- **tracker** – Trackers are arbitrary codes/strings that become associated with a user

account. Sent and recorded for new users only. You can use this for your own internal tracking.

- **anon** and **anonOnly** – When a player is playing “for fun”, you can omit playerid and sessionid and set both anon=1 and anonOnly=1. (e.g. &anon=1&anonOnly=1). Setting anon=1 defaults to an anonymous/fun session, unless a previous session is found. anonOnly=1 enforces that previous “real money” sessions are expired instead of reused. It is strongly recommended that you always use anon alongside anonOnly.

FUNCTION CALLS

This section details the breakdown of each function call with information on the data we will send. Your system may require additional parameters/data to be sent. In this case, please contact your Rival representative.

> **`validate`** Authenticates a player against your system.
>
> **Request**
>
> | | |
> |---|---|
> | `playerid` | UUID to identify the user on your system. |
> | `sessionid` | A session identifier/hash unique to the player's current active session. This identifier should only persist and be valid during an active play session and destroyed afterwards. |
> | `function` | A string, set to 'validate'. |
> | `hmac` | Hash string of request for authentication (See security, above) |
>
> **Response**
>
> | | |
> |---|---|
> | `balance` | A float representing the player's current balance. |
> | `currency` | A string representing the player's currency (e.g. USD, EUR). |
> | `error` | Details of the authentication error (optional). |

`user_class`	A string representing the player's class (e.g. Default, Default Low Limits) (optional)

Typically, a user will register and log in to your system using a login form on your site (or in your software). After a successful login, the user will choose to launch a Rival game. The game will load in an iframe which is passed the player's UUID and unique session identifier. Our gaming frontend will automatically contact the Rival Engine, and the engine will then make the `validate` call to your system, which will ensure the user is logged in and authenticated.

If an error occurs, the error property should be set and the Rival Game Engine will consider the request to have failed and ignore any other response parameters.

In addition, the response object can be appended with an optional user_class string. This user_class will affect how the Rival Game Engine handles this player's bet limits. The given user class will be permanently associated with the player. The class can be reset or changed on the next validate call. To get a list of valid class names and their associated limits, please contact Rival support.

`getbalance` Retrieves the player's current balance from your system.

Request

`playerid`	UUID to identify the user on your system.
`sessionid`	A session identifier/hash unique to the player's current active session.
`function`	A string, set to 'getbalance'.
`hmac`	Hash string of request for authentication (See security, above)

Response

`balance`	A float representing the player's current balance.
`currency`	A string representing the player's currency (e.g. USD, EUR)
`error`	A message describing the error (optional).

The Rival Engine may call the getbalance function several times during a single gaming transaction. This is required to ensure that an accurate balance is always kept between the two systems. We always rely on your system to provide an accurate player balance, and trust that the value returned in the response is available for play in our games at the instant it is called.

If an error occurs, the error property should be set and the Rival Game Engine will consider the request to have failed and ignore any other response parameters.

updatebalance Issues a credit or debit to/from the player's account.

Request

`id`	A unique request identifier, a 64-bit integer.
`playerid`	UUID to identify the user on your system.
`sessionid`	A session identifier/hash unique to the player's current active session.
`amount`	A float amount to be debited or credited to the player's account. An amount to be debited will be negative, and an amount to be credited will be positive.
`minbalance`	If this optional element is provided, the implementer is expected to assert that the player's balance is at least this amount, and to return an error if not.
`transid`	An int value which uniquely identifies the transaction. This should be used to track several debits for a single game transaction.
`gameid`	An int value. Unique game identifier pertaining to the game being played.
`rootgameid`	An int value. Unique game identifier of the root game. The root game identifier will only be available if it differs from the game identifier.
`parenttransid`	A 64-bit int value. The identifier of a related transaction from which this transaction descends.
`function`	A string, set to 'updatebalance'.
`hmac`	Hash string of request for authentication (See security, above)

Response

`balance`	A float representing the player's current balance after credit.
`currency`	A string representing the player's currency (e.g. USD, EUR)
`error`	Empty on successful update. Can be set to a (non-zero) error code or string containing details of any error.
`user_error`	Used to display a user-facing error. Must be one of the values *LIMITS_EXCEEDED* or *INSUFFICIENT_FUNDS*. These will display a message on the client indicating that a limit has been exceeded or that the player does not have sufficient balance to cover the debit.

Whenever the player performs an operation that is required to update their balance in a Rival game, an updatebalance call is issued. Typically, this occurs at the end of a gaming transaction, but could also occur several times during a pending transaction. Your system must assert that the player has at least the given minimum balance then add or subtract the given amount to/from the player's account and return the updated balance.

For example, if a player places a wager for 10 units of currency and wins 100, the system will issue an updatebalance request with an amount of 90, and a minbalance of 10. This performs the required change with a minimum number of operations while ensuring that at the time of the wager the player has sufficient funds to cover the wager.

Note: The Rival engine is quite powerful and flexible. The above-detailed JSON calls are outlined as a straight-forward method of integrating quickly and easily. If your system requires a different form of communication or adjustments made to parameters, please contact your Rival representative.

`rollback` *(optional)*	Rolls back (reverts) an updatebalance request.

Request

`id`	A unique request ID from an updatebalance call.
`playerid`	UUID to identify the user on your system.
`sessionid`	A session identifier/hash unique to the player's current active session.
`function`	A string, set to 'rollback'.
`hmac`	Hash string of request for authentication (See security, above)

Response

`balance`	A float representing the player's current balance after rollback.
`currency`	A string representing the player's currency (e.g. USD, EUR)
`error`	Empty on successful credits. Can be set to a (non-zero) error code or string containing details of any error.

In the case of a communication error when attempting an updatebalance operation, the Rival Game Engine will internally roll back the transaction and attempt to communicate the rollback with the primary platform.

In case there is a communication error during the rollback call, the state of the individual requests is maintained internally and can be reconciled manually.

Note: This method is optional. For the best possible user experience, we recommend that this method be be implemented. The Rival Game Engine does not however require that the rollback operation be implemented, as it is possible to reconcile rolled back transactions

manually later.

`status` Checks if the service responds

Request

None

Response

None

This method acts as a ping to make sure the service is operational. This method requires a status response header of '200' to be returned.

TESTING

Rival will set up a fully functional test environment in order to facilitate testing of each game, and the communication between our system and yours. In order to properly (and thoroughly) test the whole system, we will need you to provide a permanent test environment as well. The typical requirements for this are as follows:

- A test URL (hostname and path) to be used to test the above functions. This should expect calls to be made exactly as in a production environment.
- A web site or web interface to log in an launch a Rival game, loaded in an iframe, the same way it will be launched in production.
- Test accounts for your system that have sufficient balance (or adjustable balance), permissions, and experience levels in order to be able to test each game. In most cases, several (4 or 5) test accounts are necessary so our team can work concurrently on different aspects of the system.
- For initial testing (before the website or web interface becomes available), a method of creating sessions in the test environment may be required for Rival developers use in integration testing. This may be an additional “createsession” web service method or a rudimendary web UI. In other words, a method of obtaining a valid “playerid” and “sessionid” is required for Rival to begin development.

TRANSACTION AUDITS

Once the test system is up and running, you will be able to see transactions for each player both on your system and the Rival system. Since we will not have insight into how data is being stored on your system, it is highly recommended that you do a full transaction audit on each game to ensure that transactions on our system match up with transactions on yours.

This is typically done by playing a specific game and then comparing the initial balance, debits and credits appearing in the game, and the final balance on both systems to ensure there is nothing being missed by either system.

CUSTOMIZED CASHIER AND LOBBY FUNCTIONALITY

While playing a Rival game, players will be presented with options to exit the game (i.e. Return to Lobby) or visit the cashier to make new deposits. We can direct players to separate URLs as required to suit your needs. For instance, if you would like all users to return to your home page when exiting a Rival game, or if you would like them to visit your custom cashier page to make a deposit.

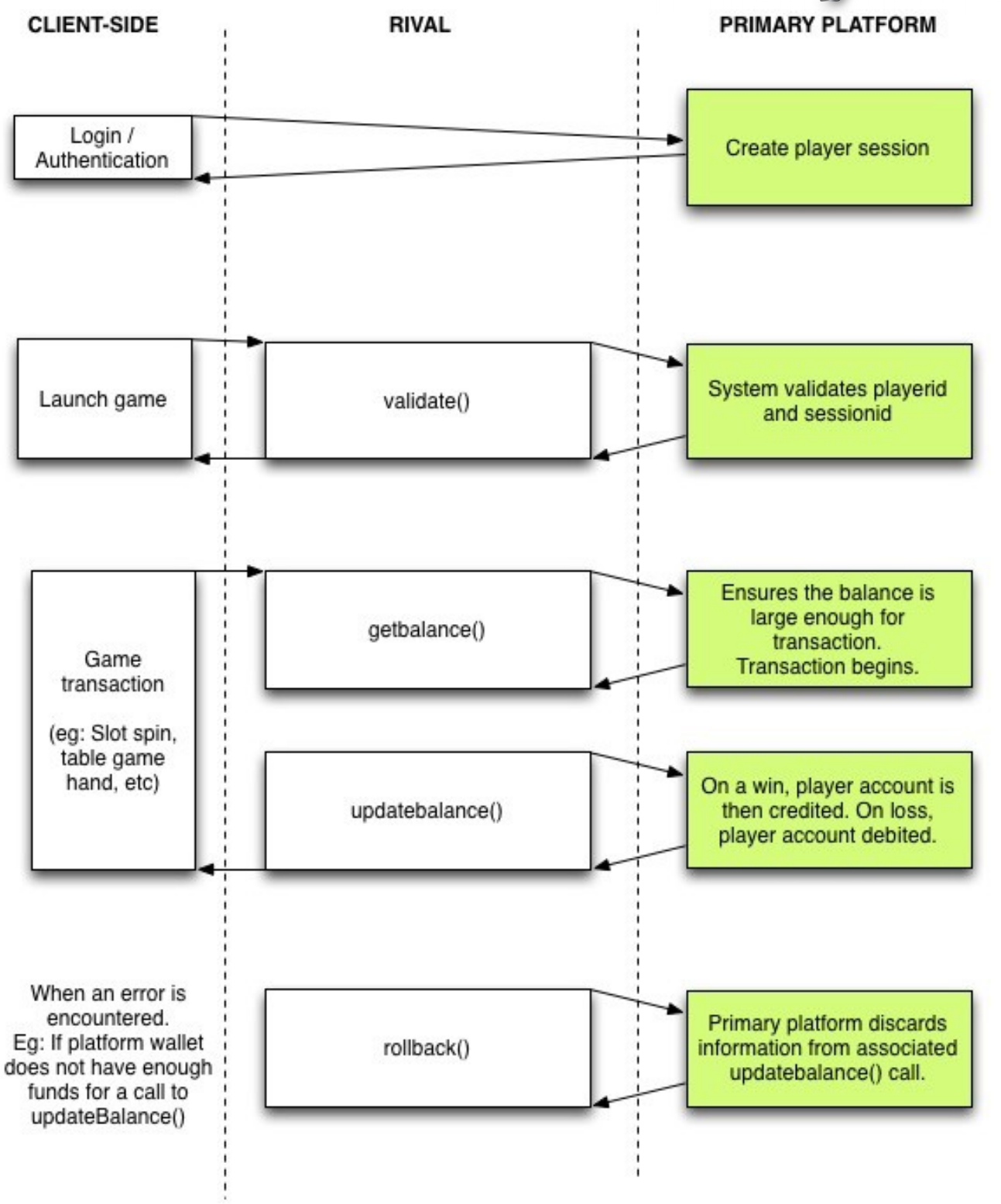
We highly recommend that you provide separate URLs for both Staging and Production environments. You may also choose to provide completely separate URLs for use with full-sized Flash games and Mobile games.

When you have decided on which URLs you would like to use, please provide our support team (support@rivalpowered.com) with details as follows:

- URL that players will be sent to when exiting a Flash game. If this URL will also be used for Mobile games, please specify this. Otherwise, let us know the URL mobile users will be sent to when exiting games.
- URL that players will be sent to when they have a low balance or choose to visit the cashier from a Flash game. If the same URL will also be used for Mobile users, be sure to indicate this. Otherwise, specify a separate URL for Mobile users.

Don't forget to specify all of the above URLs for Staging and again for Production.

OVERVIEW DIAGRAM


CLIENT-SIDE
RIVAL
PRIMARY PLATFORM
Login / Authentication
Create player session
Launch game
validate()
System validates playerid and sessionid
Game transaction
(eg: Slot spin, table game hand, etc)
getbalance()
Ensures the balance is large enough for transaction. Transaction begins.
updatebalance()
On a win, player account is then credited. On loss, player account debited.
When an error is encountered. Eg: If platform wallet does not have enough funds for a call to updateBalance()
rollback()
Primary platform discards information from associated updatebalance() call.