# GhostRunner Game Engine

**API Design Document**

**Version:** 1.0
**Last Updated:** February 27, 2017

**Author:** Alamgir Farouk

# Table of Contents

# Legal Page

**Confidentiality Agreement**

The undersigned reader acknowledges that the information provided by GhostRunner in this document is confidential; therefore, the reader agrees not to disclose it without the express written permission of GhostRunner Inc.

It is acknowledged by the reader that information to be furnished in this document is in all respects confidential in nature, other than information which is in the public domain through other means and that any disclosure or use of same by the reader may cause serious harm or damage to GhostRunner Inc.

Upon request, this document is to be immediately returned to GhostRunner.

_____
Signature

_____
Name (typed or printed)

_____
Date

# 1 OVERVIEW

This document provides the schema and API details for the Baseball Game engine.

The schema is based on Baseball rules, GhostRunner game and remote interaction requirements. We describe each table separately, and show their combined relationship at the end.

UID (User identifier) represents a user (logged in). Complete description of User entity is omitted in this document.

NOTE: Actual table names may be different based on schema requirements.

# 2 ENITITIES

These are the high level entites we will use.

## 2.1 OWNERS

An Owner is a user in the game, identified by UID and other meta data.

## 2.2 PLAYERS

This represents a player in the 'pool' of players. This table captures personal (non-game) related information about a player.

## 2.3 PLAYERSEASON

A particular player's performance during a season is caputured in this table. It is also known as the 'player card'.

## 2.4 TEAMS

One owner can have multiple teams. So, teamId is incrememented. Teams may become dormant or retired. We can decide how many active teams are possible.

## 2.5 TeamPlayers

This represents a player in the 'pool' of players. This table captures all the information about historical players, possibly using auxilliary tables for historic data.

## 2.6 TeamLineUp

A lineup is created by an owner using his team players. A lineup can have players only from a team. So the Team comes first .

## 2.7 TeamLineUpPlayers

Additional attributes are needed for a lineup such as play position etc.

## 2.8 Game

A game represents a new game being played between two TeamLineUps. At any time, the game entity records the state of the game, who is playing, who is standing where, etc. It should record all information such that a game can be reconstructed and play can continue. Current scores are also cached although they could, in theory, be computed from the playevents entries.

Games
- Home(id)
- Away(id)
- id(id)
- current inning
- player positions
- [game state]
- [ui state]
- [cached score]
- …
- Inninings
- Who is at bat
- Who is pitching
- Out status (0,1 or 2)
- Base status(0 through 7) (who where)
-
-

*Figure 1 Games table*

## 2.9 GameEvents

Each play event is a chronological record of the events that combine together to form the game. The events maintain a sequence, which is important for multiplayer distributed games.

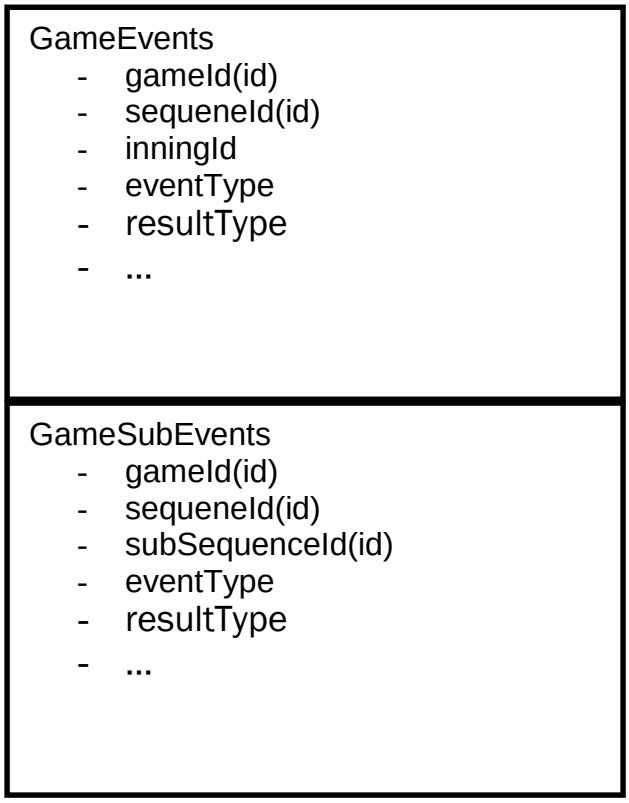


*Figure 2 BaseballGamePlays*

## 2.10 Viewers

A viewer is a user who is allowed to observe an ongoing game. A viewer-game relation table will be created (omitted as too straightforward)

# 3 DATABASE SCHEMA

The following diagram shows the overall schema of the game engine. Most attributes are omitted, only the relational model is shown.

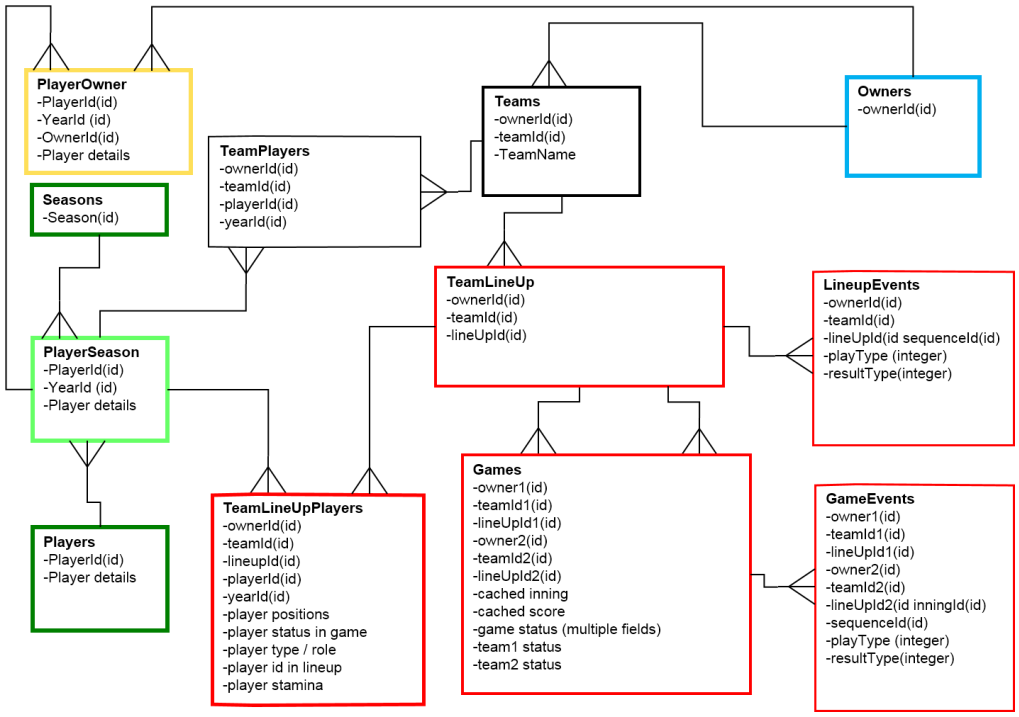


*Figure 3 ER diagram for game and teams*

# 4 GAME STATES AND TRANSITION

The two teams as well as the game itself go through unambiguous states which are persisted at any time, all transitions between them being atomic.

We omit the game states for simplicity. In general, the game state will capture the 'turn' information (Owner1, Owner2 or System), but within the turn, the actual state of team will be maintained in the TeamLineup table, and individual player states will be maintained for the appropriate player in TeamLineUpPlayer table.

## 4.1 INVITER STATES

We only consider the state transitions for the Inviter (the one who initiates the game) and the Invitee.

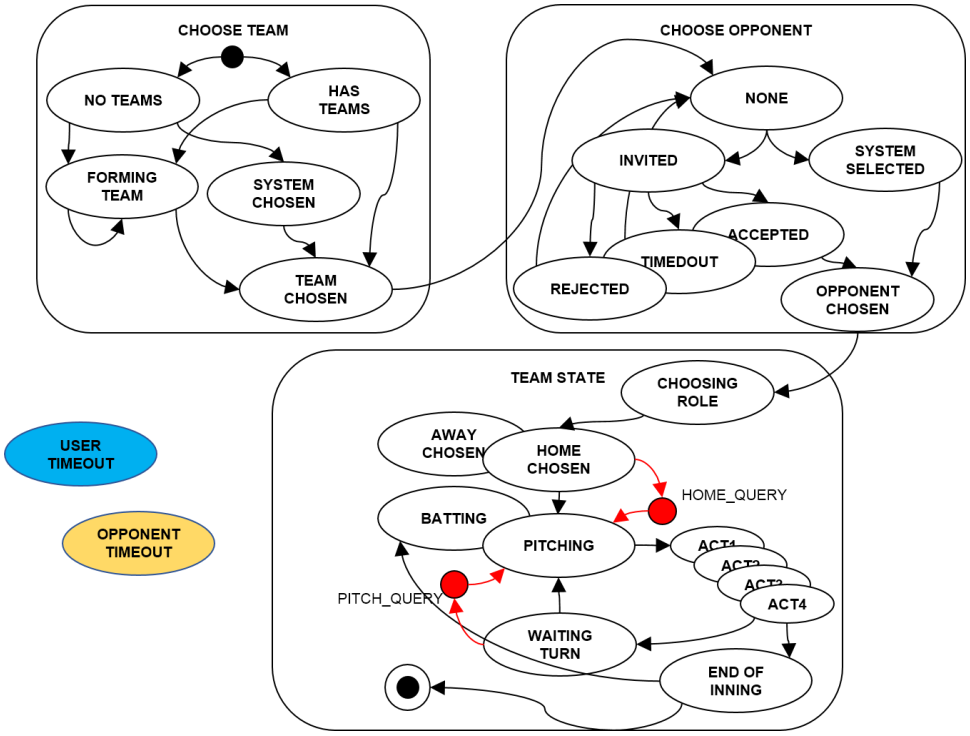


*Figure 4 Inviter (the initiator of the game) state transitions.*

## Invitee states

The person being invited goes through a slightly different set of states at the beginning, but we repeat the full life-cycle for completeness.

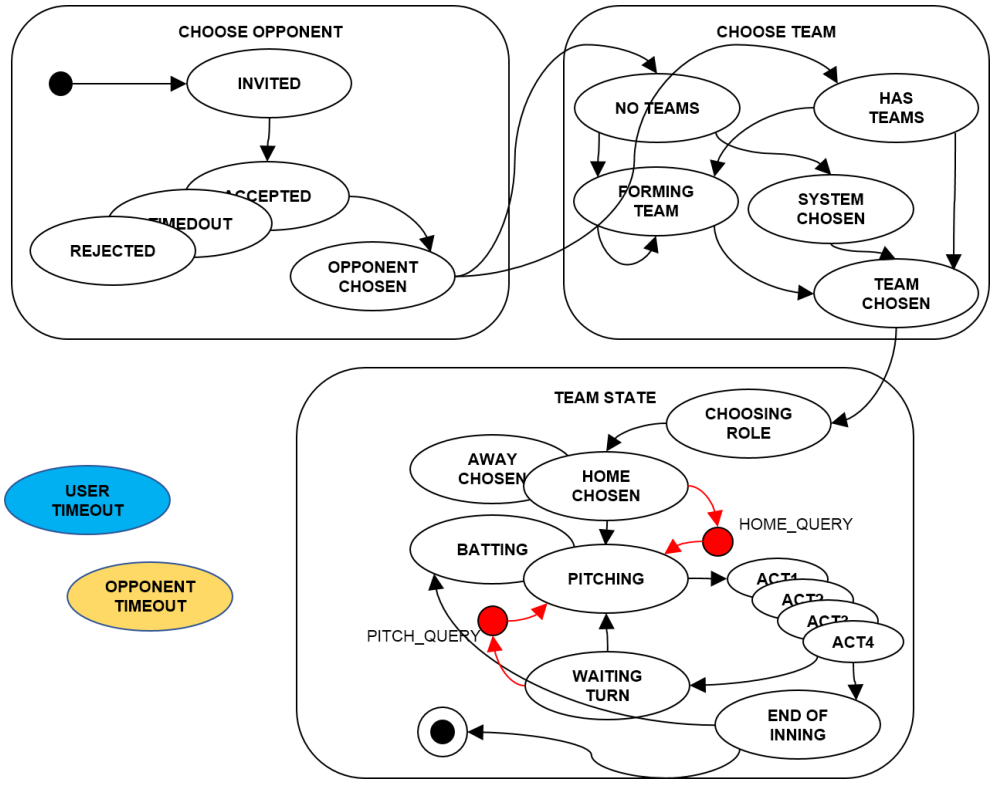


*Figure 5 Invitee (person who was invited) state transitions.*

## Game flow

The game flow itself is modeled as a flow diagram, to better illustrate the actors (owner1, owner2 or system)

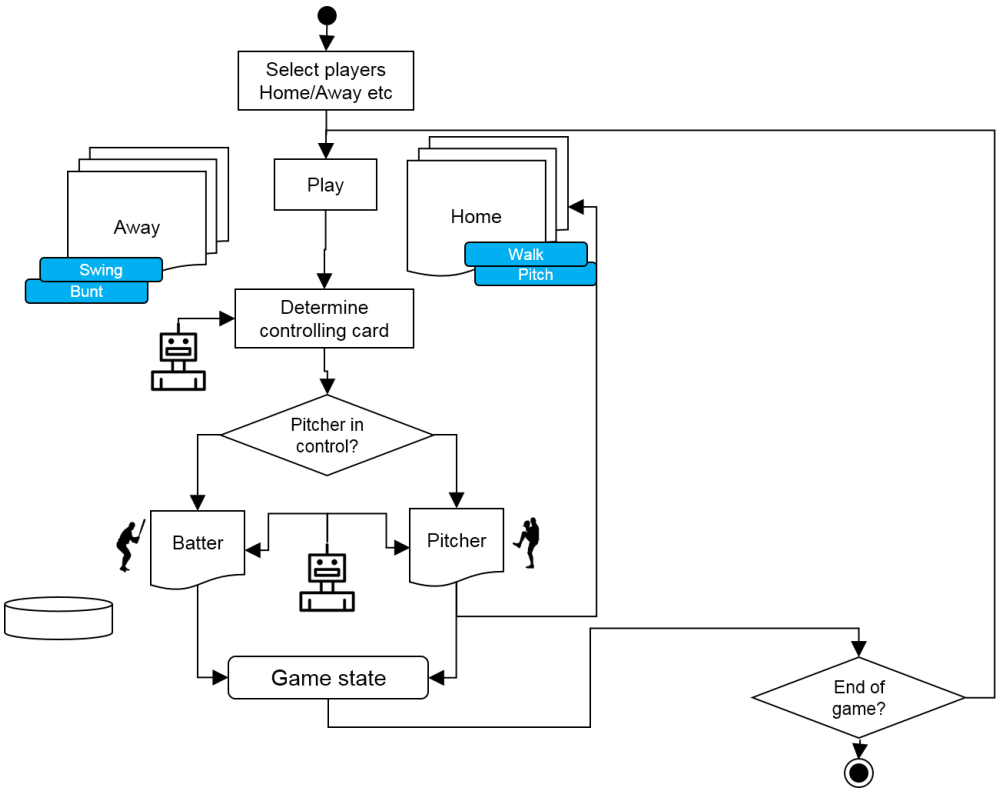


*Figure 6 Game flow diagram.*

# 5 GAME LOGIN

The login will use the standard authentication from our authentication package, with the following requirements

The login component (javascript object?) will be a separate component but it can be built as part of the package.

The login will set a client side cookie (as usual) with the UID as well as fire an event on login and logout and allow attaching listeners to it. Other pages may also want to listen to this event, so it may be a good idea not to have any dependencies other than jquery etc.

The login will maintain all user details (username, UID) by retrieving them from the server. If retrieval fails, state will be logged out, cookie removed (same as now).

The login will parse the URL and use the embedded UID and retrieve the username etc., when the URL contains a UID. Invalid UIDs (when server throws an error) will be ignored silently.

Valid UIDs will result in logged in status as usual.

# 6 GAME BROKER SIGNALING

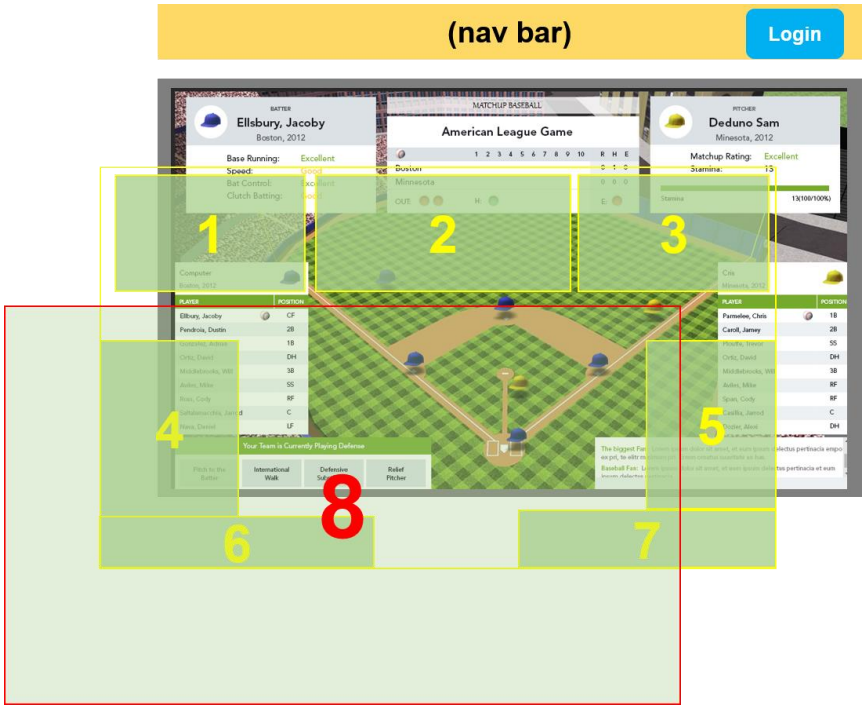


Figure 7 Main UI components

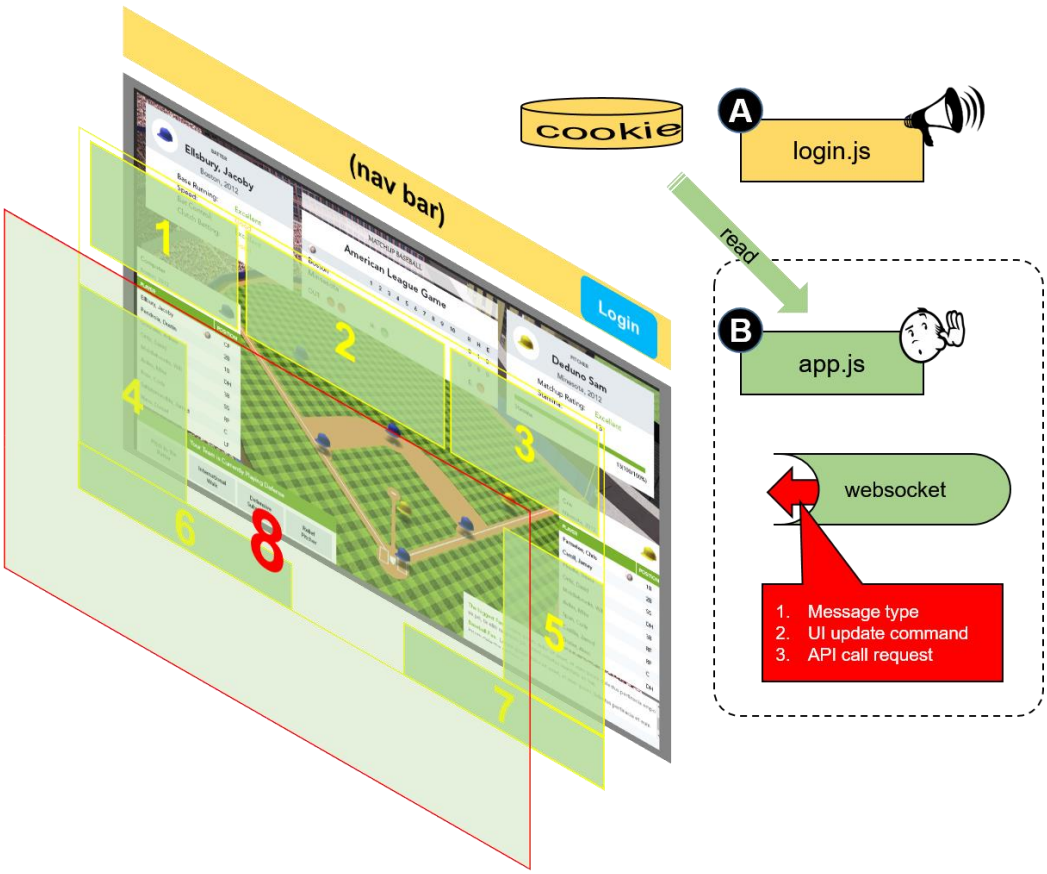


Figure 8 Signaling via websocket will control the masks and UI state. Default state will be full mask (8)

# 7 USE CASES

We cover a few important use cases.

(TBD)

# 8 API SAMPLES

This is a sample API description

***isUserNameAvailable***

| | |
|---|---|
| **REST Mapping:** | *[GET] /authentication/isUserNameAvailable/?username=username0* |
| **Description:** | *Check if the username is available or not.* |
| **Parameters:** | *username* |

| | |
|---|---|
| **Return Value:** | *Result object, success must be true.* |
| ***Exceptions:*** | *UnableToComplyException is thrown if the supplied arguments are invalid. PanicException is thrown if there is an internal server error.* |