Cubeia Poker

Software Design Overview

2010-11-29

Fredrik Johansson

Scope

The aim of this document is to provide an overview of the architecture and software design of the Cubeia Poker developed for deployment on the Firebase game server platform.

This document will not provide protocol definitions or detailed class level design.

This document will not explore the distributed, clustering features of Firebase.

Firebase

The Firebase platform is a game independent, high availability, scalable platform for multiplayer online games. It is developed from the start with the gaming industry in mind. It provides an API for game development using event-driven messaging and libraries for point-to-point client to server communication.

Cubeia Poker

Cubeia Poker is an online multiplayer poker game. The backend is written in Java and should be deployed in Firebase. The frontend is written in Action Script (Flex) and is targeted for deployment on a webserver. The back office is written in Java using the framework Wicket and is designed as a web application.

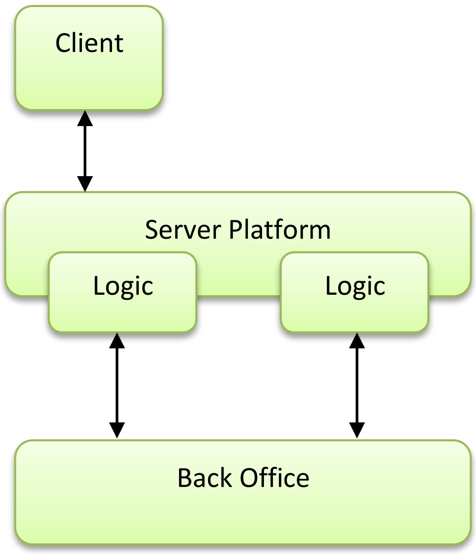
Contents

Network System

A gaming network consists of multiple functional parts. In the picture below we have isolated the different parts as:

- Client
- Server Platform
- Server Logic
- Back Office

The different parts will work together like:



There will be multiple different actors on the system but since that is not the scope of this document, they are not included in the overview.

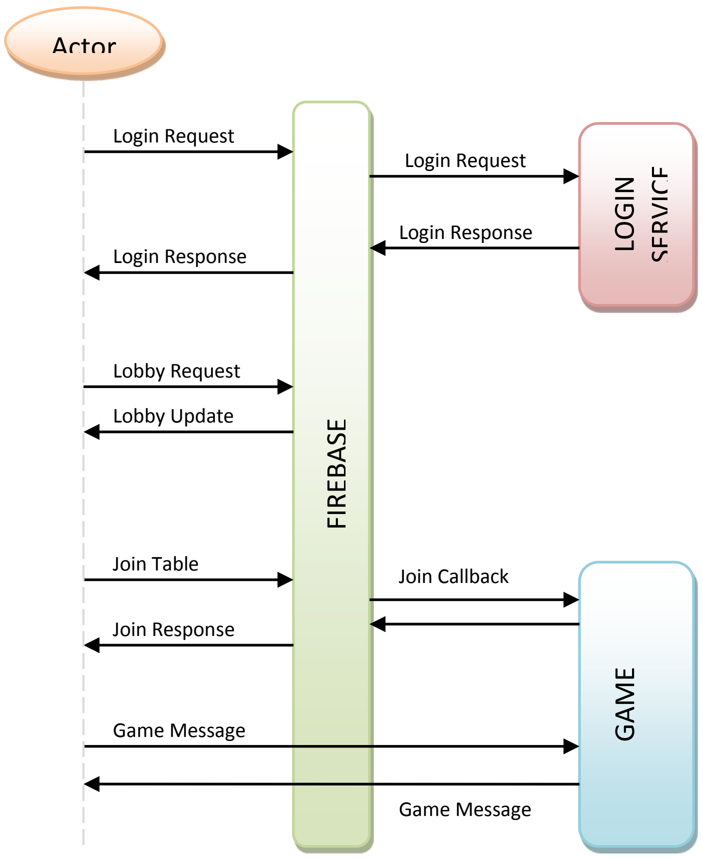
Firebase Game & Services

Games that are deployed on Firebase will use the Firebase Game API and take advantage of the functional features provided by Firebase.

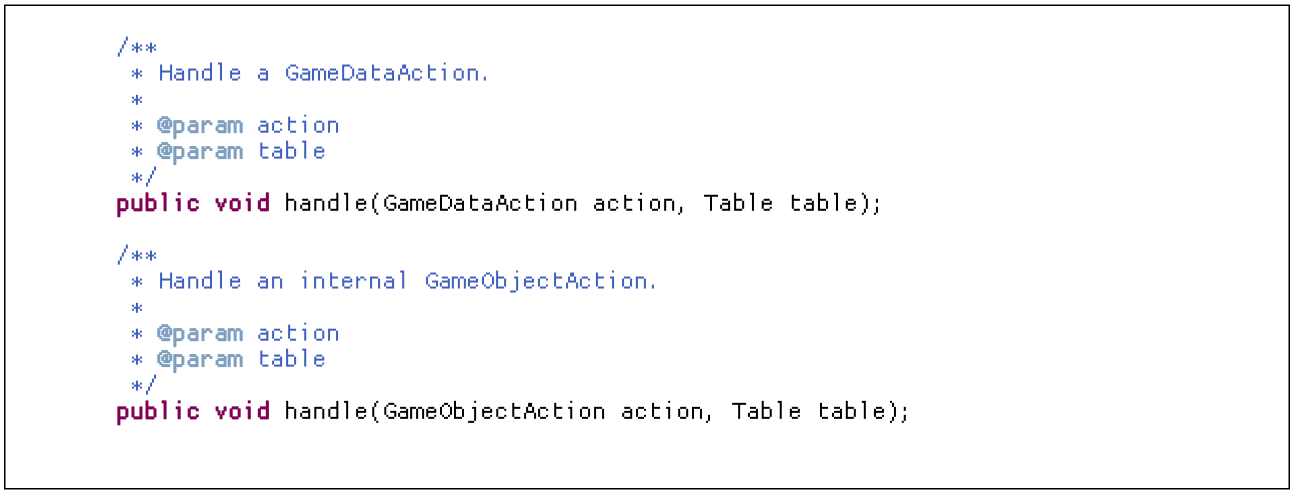
Services are generic providers of business logic that can be accessed directly through the API or addressed by the client through the designated protocol.

The complete list of functional features is described in the Firebase API documentation (Wiki and Javadoc, available at www.cubeia.org).

Some functional features can generate a call back to the implementing game or service. This is illustrated below.



The Game API consists in its minimal form of:



```
/**
 * Handle a GameDataAction.
 *
 * @param action
 * @param table
 */
public void handle(GameDataAction action, Table table);

/**
 * Handle an internal GameObjectAction.
 *
 * @param action
 * @param table
 */
public void handle(GameObjectAction action, Table table);
```

Apart from the API for handling incoming actions there are dependencies to other Firebase specific classes, e.g. table. These are mainly used for external communication and communication with deployed services.

Poker Project Structure

Server Side

The complete poker java project is divided into separate sub projects (or modules). This is according to best practices for handling dependencies. The project uses Maven 3.0 for dependency management.

In order to isolate the poker logic implementation from the Firebase/Server communication and other integrations, the basic game artifact is been divided into two separate modules:

- Poker-Game
- Poker-Logic

Isolating the logic (i.e. game rules etc) also makes unit and automated testing much simpler.

The project also has the modules listed below. Each module is handled as a separate project with separate test harnesses (where applicable) and separate dependency management.

- poker-protocol
- poker-tournament
- poker-persistence

Support modules which are not included in the game archive are defined as below.

- poker-bot

The responsibilities are described below

Guice

The Poker implementation uses support module in Firebase for Guice to take advantage of IOC. Guice is implemented by the PokerGame by extending GuiceGame. By using Guice as IOC we can inject state and/or service directly into our classes and need less code for wiring or parameter forwarding.

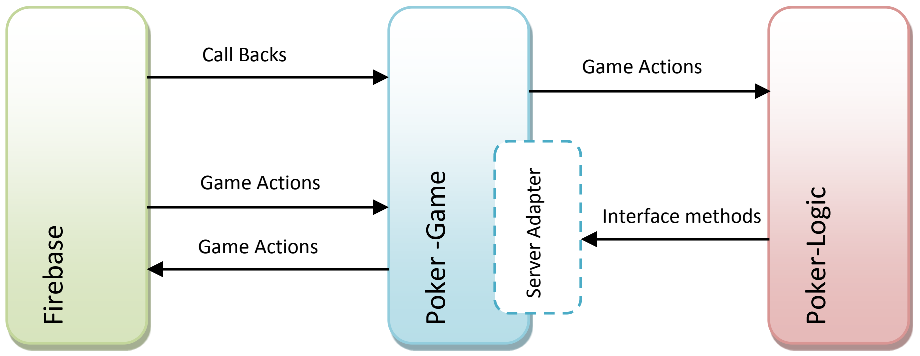
If you are planning on working with the poker codebase it is highly recommended that you understand Guice and take advantage of it.

Poker Game

The poker-game module acts as an adapter layer between the game specific logic and the server layer. This module is depending on:

- firebase-api
- poker-logic
- poker-protocol, described later
- poker-tournament , described later
- poker-persistence , described later

The poker-game maps routes incoming actions to the logic. The poker -logic layer will access the poker -game through a call back interface defined in the poker -logic.



The Poker Game module also takes on other responsibilities such as handling all external dependencies and communication, e.g. database(s), service, wallet, remote calls etc.

Additionally the Poker Game will handle scheduling of timeouts and verification of incoming actions. If the Poker Logic requests an action from a player, then the Poker Game will guarantee that the next incoming player action will be from that specific player. All out-of-order or lagged behind responses are filtered out.

Isolating all the external dependencies in the Poker Game module makes it easier to manage the dependencies and keeps the Poker Logic module clean and well defined. The server adapter layer is also very well suited for making a mock implementation for improved unit testing.

Poker Logic

The Poker Logic module contains the game rules. This is where how to play poker for a specific rule-set (e.g. Texas Holdem or Omaha) is implemented. The dependencies are:

- poker-protocol

The Poker Logic module includes the Server Adapter interface which will be used for communicating with the server implementation. The Poker Game module provides a Firebase implementation of the Server Adapter.

Example of Server Adapter interface method:

```
    /**
     * Request action from a player.
     * The Action Request object will include all contextual parameters such as
     * player, timeout, available actions etc.
     *
     * @param request
     */
    public void requestAction(ActionRequest request);
```

The interface method above will send a request to the server adapter layer (i.e. Poker Game when deployed) to send an action request to a player.

The poker logic should not depend on any server implementation classes. If the logic needs to access a specific server or service implementation object then that needs to be wrapped in the Server Adapter or a new Adapter interface defined in the Poker Logic module. Keeping external dependencies out of the poker logic module ensures low coupling and high cohesion, thus providing high testability.

Poker Protocol

This module holds the definition of the poker communication. The Poker Protocol module uses the Firebase Styx wire protocol to generate packets. Below is an example of a defined packet:

```
<!--
  Request action from a player.
  This is sent to all players so everyone is notified about who is supposed to act.
-->
<struct name="request_action">
  <var name="seq" type="int32"/>
  <var name="player" type="int32"/>
  <list name="allowed_actions" type="player_action"/>
  <var name="time_to_act" type="int32"/>
</struct>
```

The packet above is sent to a player requesting him/her to act.

The Styx wire protocol will generate packet definitions and factory objects for:

- Java
- C++
- Flash

Poker Tournament

This module implements the Firebase tournament API for running poker specific tournaments. This module currently supports sit and goes style tournaments only.

Poker Persistence

This module contains entity objects for persisting data to a database. The Poker Persistence artifact follows the Java Persistence API (JPA) specifications and is deployable directly in Firebase.

The JPA specification is a standardized ORM framework. Firebase uses Hibernate internally as JPA provider.

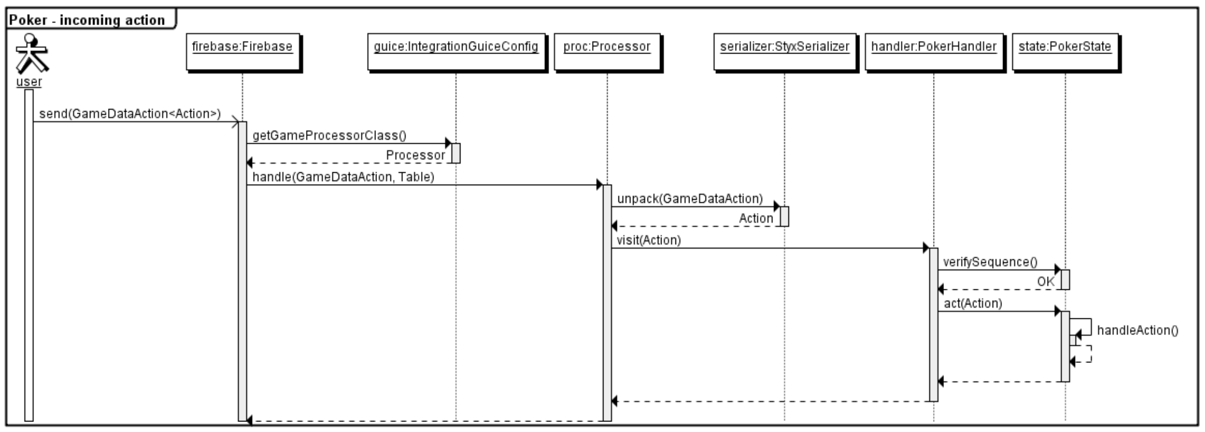
Sequences

Select use cases are described by sequence diagrams below.

Player Action Performed (Game Module)

Initial State

A player is seated at a table and game play has started. The game has requested the player to act and the player send his action. The action reaches the server before any timeout is triggered.



Result

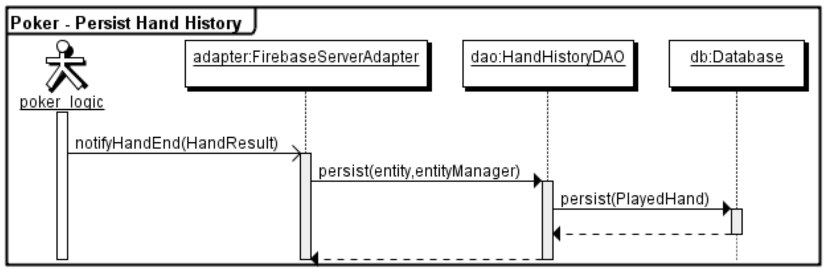
The player's action is executed in the game logic. The state is updated to reflect the changes. The player gets a confirmation of the executed action and the other players are notified of the executed action.

Persist Hand History

Initial state

A complete hand has been played at a table, the logic layer reports the end of the hand to the game layer which then will persist the accumulated hand's history to the database.

Sequence


Poker - Persist Hand History
poker_logic
adapter:FirebaseServerAdapter
dao:HandHistoryDAO
db:Database
notifyHandEnd(HandResult)
persist(entity,entityManager)
persist(PlayedHand)

Result

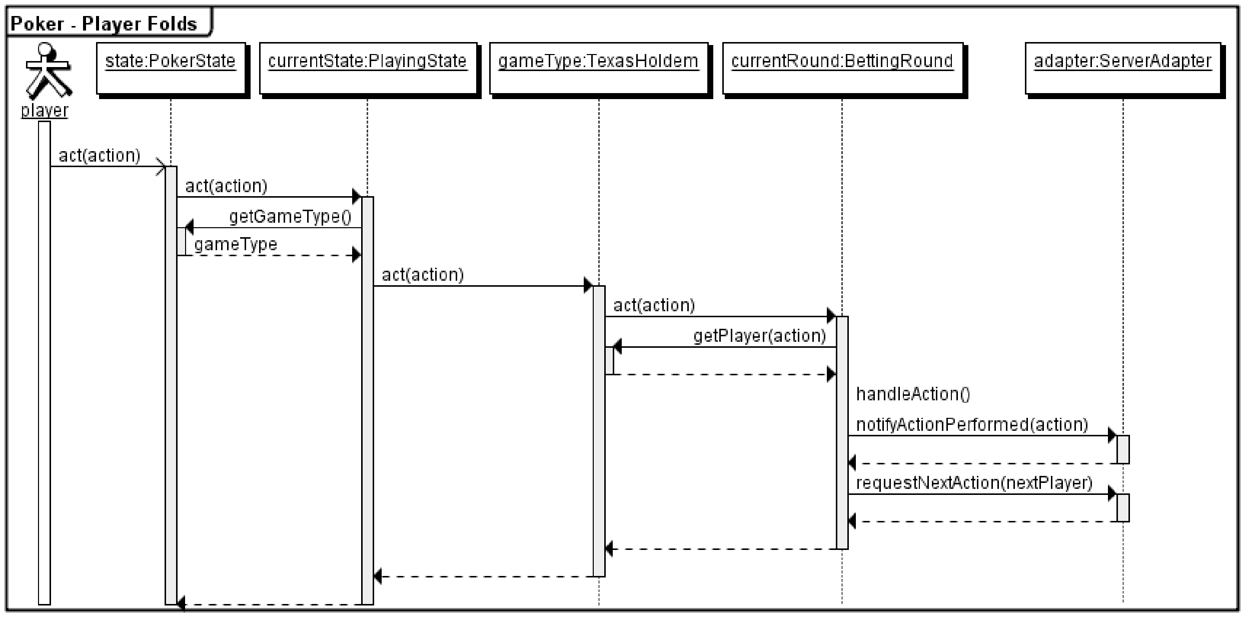
The complete records of the hand are written to the database tier.

Player Action Performed (Logic Module)

Initial State

A player is seated at a table and game play has started. The game has requested the player to act and the player send his action. The action reaches the server before any timeout is triggered.

Sequence



Result

The player's action is executed in the game logic. The logic module will update state and notify the adapter accordingly.

Developer Guidelines

This section is a guide line to the development goals and conventions for Cubeia Poker. These are guidelines only if you are developing on Cubeia Poker for internal use, but if you are intending on contributing then you must follow these guidelines.

Server Side

Decoupled Server

The Firebase server should always be able to run the Poker game without any connections to external services. Having dependencies to external servers/databases will increase complexity of setup as well as startup times. Databases should be optional (see current poker database implementation), if you really need a database then consider starting a HSQL or other in-memory/file based database is there is no external database configured.

The same goes for external services, if there are external services then they should be able to be turned off. If you have a wallet service for instance which is undoubtedly needed for bringing in cash to a poker table then you should also provide a mock service which can be deployed locally in Firebase. See the Cubeia wallet implementation used in the Poker as a working example.

This may seem overly strict and perhaps your use cases has high coupling to stored data. However, complexity across the system tends to grow and new integration points will not increase complexity linearly but exponentially. We do not need another entangled spaghetti system so keep integrations clean and decoupled.

Integrations / External Dependencies

This one goes hand in hand with Decoupled Server. All remote dependencies, integrations and/or company specific integrations must be isolated in services and served through a generic adapter interface.

Example, let's say you want to post every winner in a hand on a JMS. DO NOT implement a hook to the JMS in the poker logic or poker game code directly. What you *should* do is to:

1. Make an adapter that has poker domain specific methods
2. Make a service that handles the JMS calls

Then you can wire up your adapter through configuration or check if the service is deployed etc. The key to remember is to keep the poker source code clean of specific business requirements not directly related to poker and isolate them into services.

State

Remember that all attributes in PokerState will be serialized and sent over the LAN when running a cluster for fail over purposes. Keep the state small and make sure it is serializable!

Unit Tests

Apply unit tests wherever possible. This is usually hard in integration boot strapping, but should be certainly doable within the poker logic and other isolated components.

Admin Web

General

The admin web application for poker is built in Wicket. The idea is to handle poker specifics only. If you are using Cubeia Network then you can integrate the admin pages through an external link, this will still require separate authentications though.

Client

The client has been updated to Flex 4.

Web/AIR

The idea of the Action Script client is to be able to render it to a web client and an AIR client using mostly the same code base, if this is entirely possible or not is still to be seen. However, by building SWC's from the existing projects and having two thin render projects (i.e. Flex and AIR) it might be doable.

Skinnable

The original project used Flex 3 and the skinning options provided. This is not backwards compatible in Flex 4, so the skinning logic is broken. The idea is to provide skinning by downloading external CSS and resources from a 3rd party web server. This way the rendering process does not have to be done once per operator skin.