Firebase Server Developer Reference

Firebase Version: 1.9

Firebase Server Developer Reference 1.9

Introduction

Welcome to the Cubeia Firebase Server Developer Reference. This manual is intended for system developers and architects. It does not address administration or configuration of a Firebase cluster.

About Cubeia

Cubeia Ltd is a distributed systems expert company, registered in England and operating through our office in Stockholm, Sweden. We provide scalable, high availability systems and consultation based on our long experience in the gambling and Internet application industry.

Our main product, Firebase, is a game agnostic, high availability, scalable platform for multiplayer online games. It is developed by Cubeia Ltd and was built from the start with the gaming industry in mind. It provides a simple API for game development using event-driven messaging and libraries for point-to-point client to server communication. Firebase is a server platform for developing and running online games. It scales from small installations to extremely large, it is built to stand up to hard traffic and can be used for almost any type of game.

Contact

For further information, please contact Cubeia Ltd UK Filial in Stockholm. Bugs should be reported to the Cubeia online support forums. If you do not have access to these forums please contact Cubeia Ltd at the address below.

Cubeia Ltd, UK Fillial
Katarinavägen 19, 4tr
11645 Stockholm
Sweden

Email: info (at the cubeia domain)

Corporate Homepage: http://www.cubeia.com
Community Community: http://www.cubeia.org

Table Of Content

System Overview

This section briefly details the Firebase system and its design. It is intended as a high level introduction to the concepts involved. Understanding the fundamental design of the system is crucial in building reliable, high-performance applications.

Firebase comes in several different distributions. Firebase Community Edition is open source and a single server edition. Firebase Enterprise Edition adds full clustering and failover support.

Introduction

A Firebase cluster is made up of one or more independent servers communicating and sharing load in what is called the cluster topology. The following vocabulary is used when describing a Firebase cluster.

A *server* is a single instance of Firebase running inside a Java virtual machine. Normally there will be one server per physical machine in a cluster but it is quite possible to run several virtual machine on a single computer and thus simulating a full cluster.

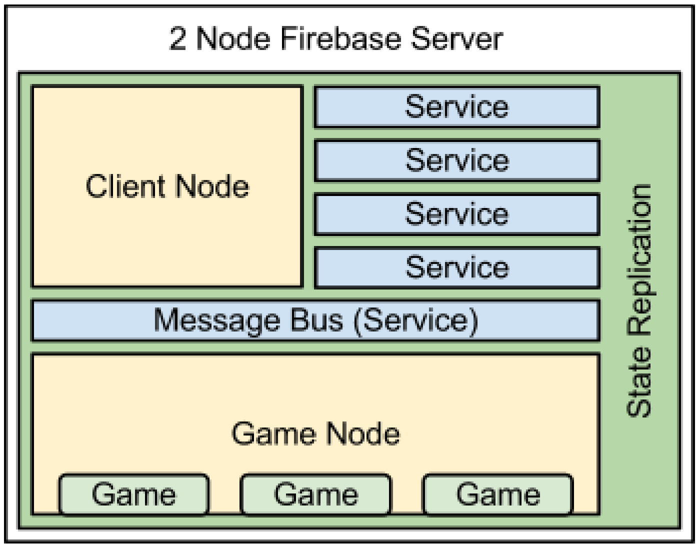
The server is made up of two main components, *services* and *nodes*.

A *service* is an internal singleton module which is available publicly in the server. Services forms the backbone of the server capabilities. Internal services are used by the system and public services can be custom developed to support games. The set of services available on a server is called the *service stack*.

Each server is capable of supporting one or more *nodes*. Whereas services are static and have their lifetime bound to their server, nodes are dynamic, distributed and can be started and stopped independently of the server. Currently the following node types are used:

- “client” - Client nodes handle client connections and session management.
- “game” - Game nodes are responsible for game deployments and actions.
- “master” - Master nodes control the cluster topology and manages kept running for the server lifetime.
- “mtt” - Tournament nodes (Multi Table Tournament) controls tournaments.

When a server is started in so called "singleton" mode, all nodes are loaded on one server at the same time making it possible for Firebase to be run on a single machine. But nodes can be combined in any manner on a server, and larger systems one or two nodes per server is the recommended topology.



A *game* is a module loaded by the *game node*s which handles the action for a particular game in the system. It co-operates with Firebase in maintaining a set of *tables* where clients can "sit", "leave" and act. The list of tables for each game is called a *lobby*, and is a shared data structure within the Firebase system.

Games and Services

The demarcation between *services* and *games* must be understood. Games are modules loaded by the game nodes when the nodes are started. And as game nodes are dynamic and can be started, stopped and moved within an already running cluster, the games themselves do not have a fixed lifetime more than as a consequence of the game nodes. Also, games only exist on servers which have one or more game nodes loaded.

Services on the other hand are loaded before all nodes, they are guaranteed to be treated as singletons, never more than one instance per virtual machine, they start and stop together with the server itself and they must be thread safe.

Services can be written to support games with common operations. For example, two games may share their accounting operation by accessing a service to handle all transactions.

Games are *isolated* which means that they will reside in their own class loader. Thus they will be separated from the rest of the system which improves maintainability and stability. Services are also isolated but have the ability to export classes, which makes them visible to the rest of the server, and through them interact with the rest of the system.

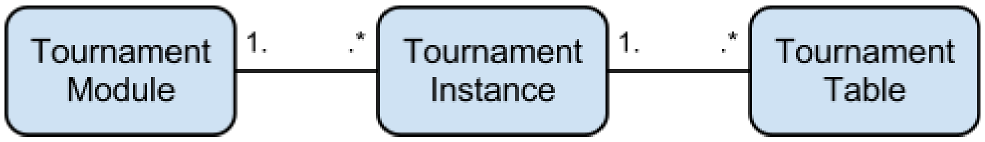
The system makes no guarantee about game class instantiation, it may instantiate any game one more more times, and may keep multiple instances alive at the same time. Services on the other hand are always loaded as singletons.

Tournaments

A tournament module are loaded by tournament nodes, and contains logic for a particular tournament *type*. As far as Firebase is concerned a tournament is not specific to any particular game, but one tournament module can potentially manage tournaments for many different games.

Just as a *game* contains the logic for a particular type of game rules, a *tournament* is the logic for a particular type of tournament rules. These can be shootouts, step by step or time limited or variations thereof.

Each tournament module has the ability to create tournament instances. A tournament instance appears in the lobby, players can register to it, and it can create and manage a set of tables.



The game play in a tournament instance takes place at its tables. These are managed by the game nodes as opposed to the tournament nodes.

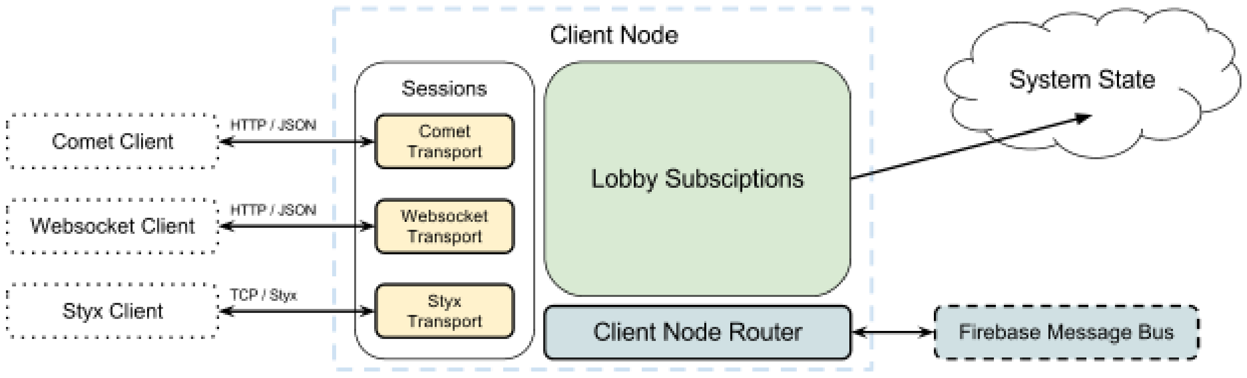
Nodes

This section will briefly present the different nodes available in a Firebase server. As defined in the Introduction, nodes are containers started within one or more Firebase service, where each nodes carries different responsibilities.

- Client nodes manages client connections and their related state, and is also responsible for the game lobby data subscriptions.
- Game nodes loads deployed games and handles game play at a subset of all available tables. In a cluster Firebase will distribute tables across all available game nodes.
- Tournament nodes loads deployed tournaments and handles tournament logic. Game play still takes place at game nodes.
- The master node is responsible for the cluster topology and acts as a controller when Firebase is deployed as a cluster (see Singleton, Clustering and Topology for more details).

Client Nodes

A client node manages connections to clients, it translates incoming events from the transport encoding to the internal Firebase actions and also manages the lobby subscriptions.



Firebase supports multiple connections types via its separation of transport from session and agnostic data encodings.

Transport & Encoding

Each client connects to a client node *transport*. Currently Firebase supports 3 different transports:

- Comet - This transport implements the Bayeux Protocol and uses JSON as transport encoding. It is recommended for legacy web clients.
- Websocket - This is an HTTP WebSocket implementation which uses JSON as transport encoding. It is recommended for state of the art web clients and devices.
- Styx - This is the original transport using Styx binary packaging over persistent TCP connections. It is recommended for thick clients or applications where latency is critical.

Sessions & Routing

Incoming messages are translated between the transport encoding and the internal action representation and then forwarded via a router to the message bus. This routing is topology agnostic and it is up to the message bus to figure out what physical destination a given action has.

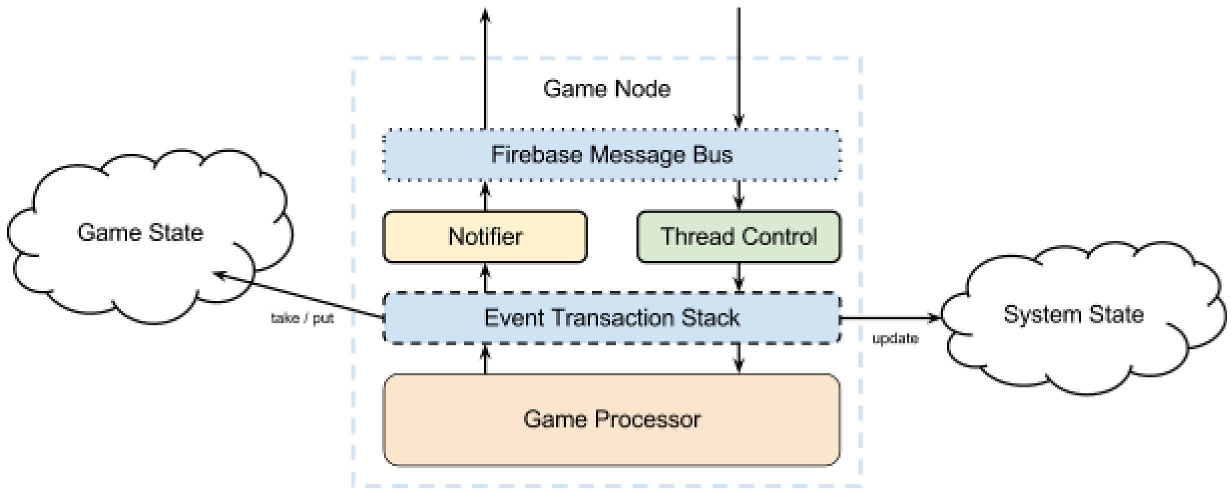
The session management also takes care of player states when clients login, logout, disconnects, reconnects etc. This involved notifying any tables where the player is sitting and updating relevant internal system registries.

Lobby Subscriptions

The game node also manages lobby subscriptions. The system state contains the lobby data for an entire Firebase installation and the client node translates lobby changes into delta changes for subscribing clients.

Game Nodes

The game nodes manages game play. As tables are created for games, they are distributed across all available game nodes for execution, and the game node will receive actions for a table, manage single threaded access to tables, update the game state and system state after the execution and also forward any outgoing events to the message bus.



Thread Control

Event in Firebase are guaranteed to be executed one at the time per table. For any given game there will be multiple tables available at all times, and the game node makes sure that incoming events are executed sequentially, one at the time, per table. This way a game programmer does not have to deal with concurrency.

Event Transaction

For each event there is a transaction in Firebase to manage flushing of outgoing events, updating of system and game states etc.

- Internal transaction - Firebase maintains an internal transaction at all times. A game developer can attach resources to this transaction via the `SystemTransactionProvider`.
- User transaction - If Firebase is configured with JTA enabled a user transaction will be wrapped within the internal transaction, but before the event is executed.

Firebase attaches resources to the internal transaction to flush the notifier, update game and system state when an event has been executed. This transaction will be rolled back if the game throws a runtime exception.

Game Processor

The game processor is created by the game developer and is called to execute a specific event on a given table. This will be discussed in detail in the server game development section, but it can be worth re-iterating that at this point there is a transaction wrapping the execution, and that there will only ever be one event per any given table executed at the same time.

Notifier

The notifier is a small transaction aware utility that forwards outgoing events to connected clients, by way of the message bus. The game processor uses the notifier to send events to player around the table.

Tournament Nodes

Tournament nodes works precisely as game nodes but manages tournaments instead of games. They are designed the same way and contain the same components:

- Execution on a given tournament is always single threaded.
- Each event for a tournament instance will be executed within a transaction.
- The tournament processor takes care of the tournament logic.
- A notifier is used to send events to clients and to participating tables.

Master Nodes

The master node manages the cluster topology. It has no other responsibility than to keep track of cluster members and coordinate halt and resume actions when members appear in a cluster or drops.

A master node is always the first node to start in Firebase, and if Firebase is run as a cluster, the first server started must contain a master node.

Game and tournament nodes are clustered by shared state, and participate in the execution of events. The master node does not, and it's clustering mode is that of a master/slave representation: there may at any time be more than one master node in a cluster, but only one will be active and act as a coordinator.

Coordinators

For each type of node - game, tournament, client or master, there is at all times one node acting as a coordinator for the type. The coordinator is responsible for actions that can only take place at one node within a cluster, for example:

- The coordinator node for game and tournament nodes handles game activation and tournament activation respectively.
- The master node coordinator act as the “master master node” and is the only actor, all other master nodes are passive.

The coordinator role is passed to the first node of a given type which appears in a cluster. Should a coordinator node be removed from a cluster, another node of the same type will be “promoted” to coordinator and take on its responsibilities.

Singleton, Clustering and Topology

When a Firebase server is started with all nodes in the same Java Virtual Machine, it is said to be running as a “singleton”. This is what would be used for development and for small installations.

When using Firebase Enterprise Edition several Firebase servers can be clustered together. They can still be running as singletons, but for larger installations a single server will normally carry one or two nodes only.

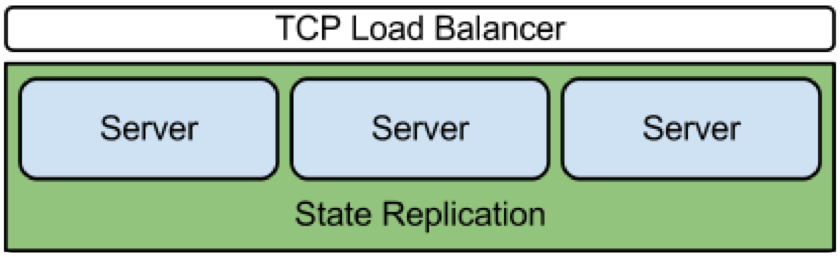
Below are some examples of cluster topologies for Firebase:

- 1 server: This server must by necessity be a singleton.
- 2 servers: Normally this topology will contain two singleton configurations to provide full fail-over support.
- 4 servers: This is where different servers takes on different roles. A lot of the internal testing at Cubeia uses 2 servers configured with master and client nodes, and 2 servers with game and tournament nodes.

Although it is possible to deploy a large cluster using singleton servers only, it is recommended to split responsibility across the cluster for increased efficiency and separation of concerns.

Events and State

A Firebase cluster balances traffic between nodes and provides transparent failover for clients and games. In order to do this it enforces a strong separation between events and state, between logic and memory.

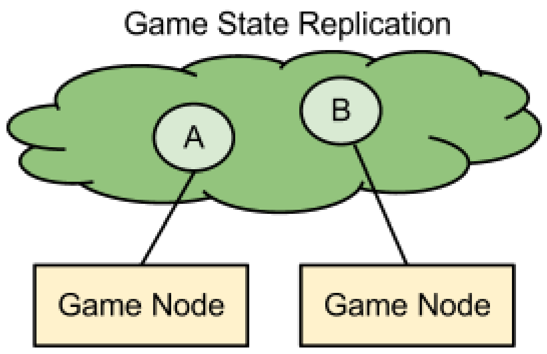


The state of a particular game is always associated with the "table" where the game takes place. The table is somewhat of a misnomer and should be thought of as an "area". Players leave and join the area as they enter and exit the game. The state of the game is kept as a serializable object associated with the table.

As a client acts on a table, the game object will be given the table and the event for processing. In other words, the game will not keep references to individual tables or individual game states, these will be handed to the game when an action takes place. In this manner Firebase will be able to balance load between tables and games transparently across several servers and network boundaries. As such, a Firebase cluster may be viewed as a kind of event-driven state machine.

Game Data Replication

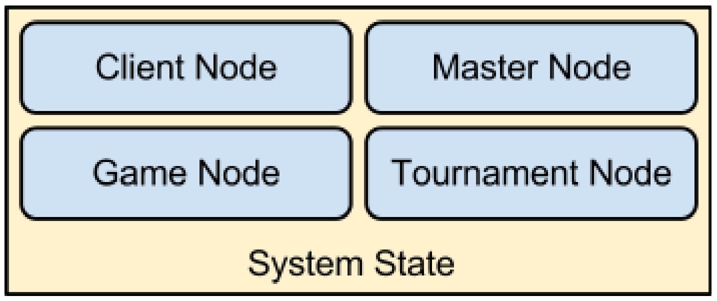
A Firebase Enterprise Edition server maintains a “game object space” as a service. It is a distributed cache which is shared between the game nodes in order to replicate game data. Each game will at any given point exist in the memory space of at least two servers for each cluster in order to provide failover should one server fail unexpectedly.



System State

The System State is a loosely distributed state shared by the entire server for the lobby and transient player data. This distributed cache is replicated between the servser asynchronously in batches.

- The game state is shared between *game nodes* only. It is synchronously updated and contains at any time two copies of each game state, spread across the available server in a buddy replication algorithm.
- The system state is shared between *all nodes* in a cluster. It updates in asynchronously batches and is used for state information, lobby data and internal meta data.



Lobby

A subset of the system state is the lobby data. Its a hierarchical data structure controlled partly by the game developer with table and game information. Attached clients *subscribe* to the entirety or a subset of a lobby tree for a specific game, and will get updates for tables in its subscription area as delta changes.

Message Bus

The message bus is the event layer binding nodes together in a cluster. It contains several channels for distributing events to game nodes, to services, or to client nodes. It is also the component responsible for maintaining the *cluster topography*, which is current layout of the cluster.

Currently there is no way of manually controlling the topography for a cluster. The *master node* will manage this by spreading all available tables and tournaments across all available nodes evenly.

Replication and Failover

These sections describes the replication process and how the system handles server down if failover is enabled. Failover is only supported in the Enterprise Edition.

> **Note:** The main difference between Firebase Enterprise Edition (EE) and Community Edition is that of clustering. Firebase EE fully supports clustering and failover whereas CE is a single server instance only.

Master Nodes

Each master node keeps a copy of the current cluster topology. All cluster changes are registered in all masters, but only the primary master, the coordinator, will attempt to resolve changes.

When a master node starts, one of two things happens:

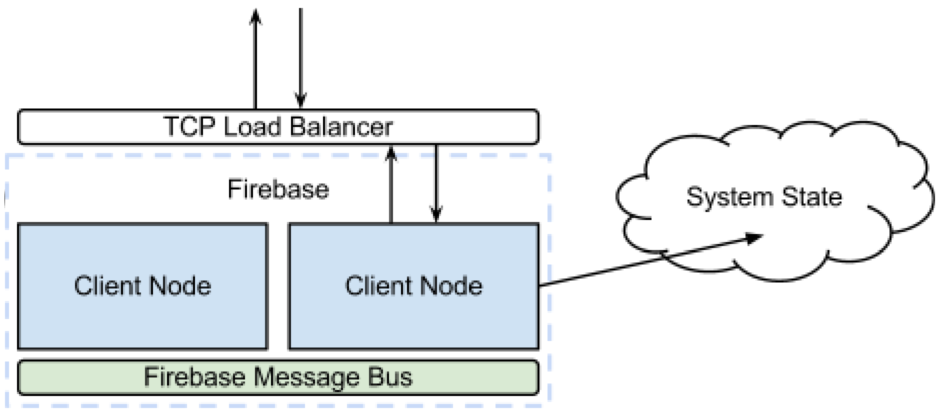
- The master node is the first master in the cluster, it will assign itself as the coordinator and resume execution normally.
- The master node is not the first master, it will mark itself as a “slave” and ask the coordinator for the current cluster state, when the state is received execution will continue normally.

Other than the state sharing between the primary master and any secondary masters on startup, no special measures are in place, and master nodes are treated as any other node.

When a node disappears from a cluster all secondary masters will start by checking if the node that disappeared was the primary master, and in such case, resolve which of the resulting master nodes will assume the coordinator role. This is resolved in “network order”, ie. by preference of age within the cluster.

Client Nodes

Client nodes are semi-stateless and do not require clustering as they rely on the system state to keep track of client sessions.



A load balancer should be kept in front of the client nodes to distribute incoming connections and it should be made “sticky” in order to minimize load on the client nodes when a client reconnects. This is especially important for HTTP based clients.

Game and Tournament Nodes

This section describes the replication and failover process for game nodes. It also applies for tournament nodes, the only change is that the game object used (i.e. the actor) is a tournament object instead of a table object.

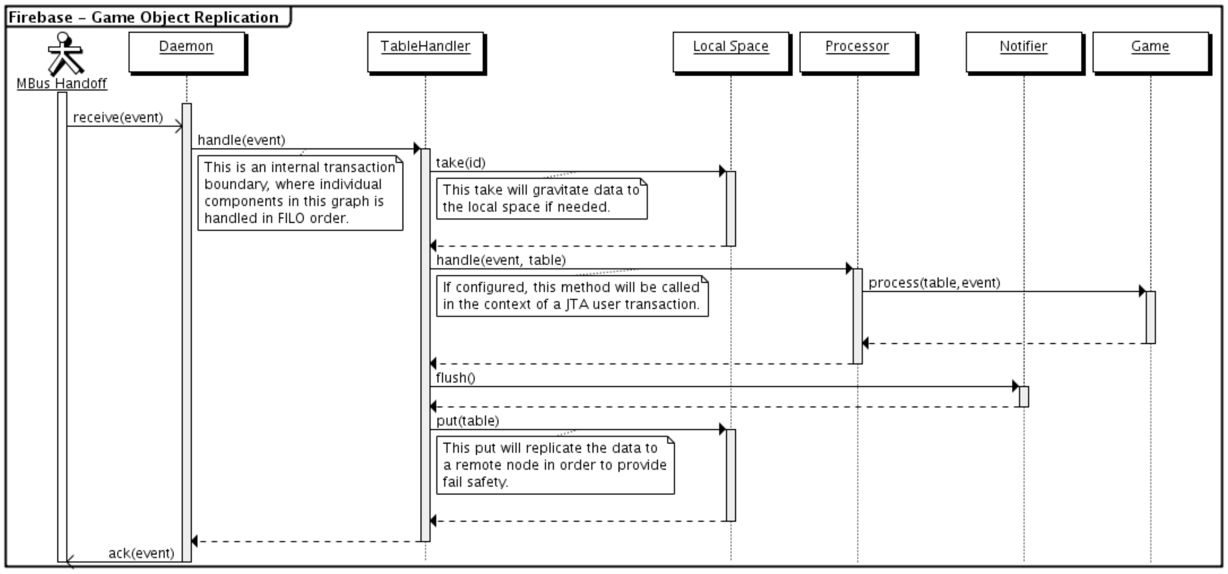
Game Object Space

The system uses a replicated space solution for replicating state between servers. The game object space exposes a very small interface heavily influenced by the Java Space API. Below is a quick overview of those methods. All object are references by a globally unique ID, in other words table ID and tournament ID.

- `get(int)` - Will return an object from the space. Acquires a read lock.
- `take(int)` - Will return an object from the space. Acquires a write lock which is kept until the object is returned to the space via `put`.
- `put(object)` - Returns an object to the space. This will be done with a write lock. If there is already a write lock (from `take`) in place it will be released.

Replication

An object will be replicated between servers when an object is `put` to the space. For regular table and tournament executions this is when the event handling has been completed and the state is returned to the space.



This is an internal transaction boundary, where individual components in this graph is handled in FILO order.

This take will gravitate data to the local space if needed.

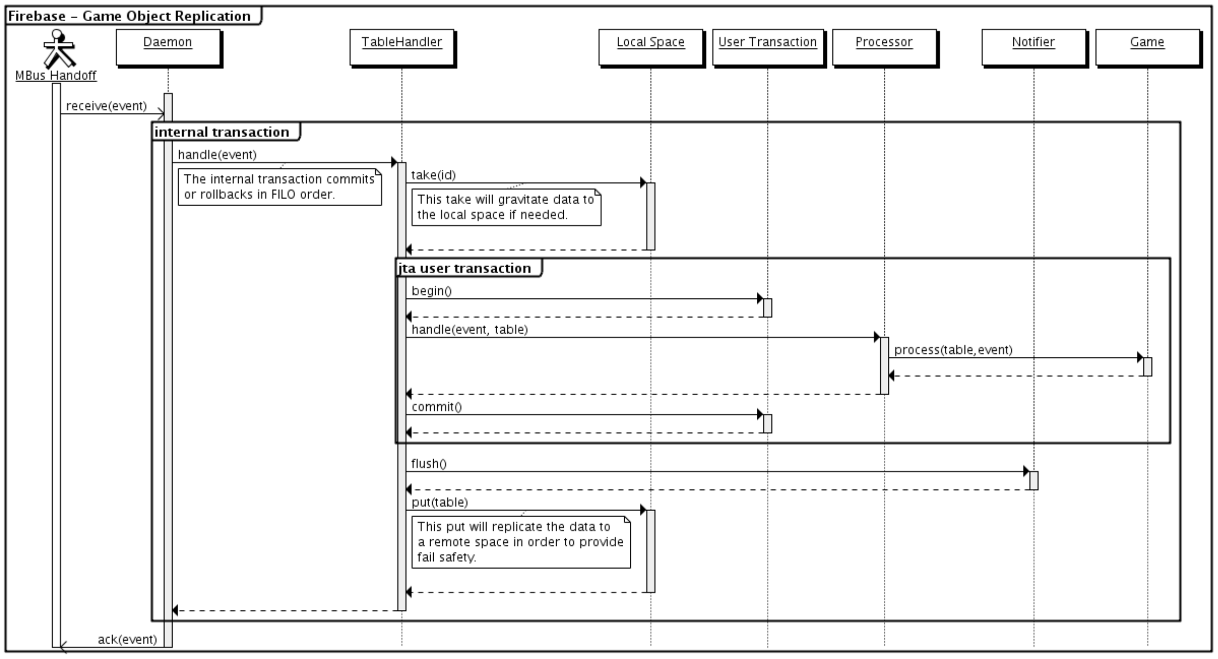
If configured, this method will be called in the context of a JTA user transaction.

This put will replicate the data to a remote node in order to provide fail safety.

Transactions

When accessing a game object with a `take` or `put` it will always be accessed it within a transaction. Thus, the execution of events on game objects or tournaments are also always performed within a transaction.

Firebase supports two different types of transactions to be used. The default mode is an internal transaction that will not be visible to the game implementation. Optionally Firebase can add a JTA transaction manager which will provide an XA transaction that the game implementation can use.

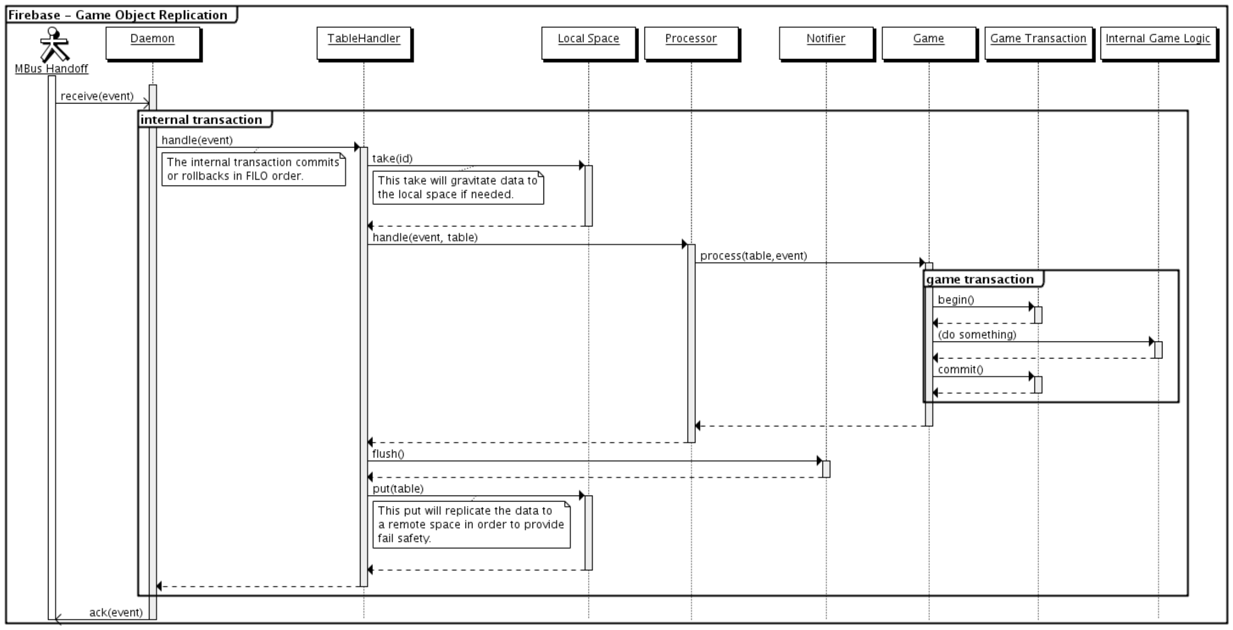

Firebase – Game Object Replication
MBus Handoff
Daemon
TableHandler
Local Space
User Transaction
Processor
Notifier
Game
receive(event)
internal transaction
handle(event)
The internal transaction commits or rollbacks in FILO order.
take(id)
This take will gravitate data to the local space if needed.
jta user transaction
begin()
handle(event, table)
process(table,event)
commit()
flush()
put(table)
This put will replicate the data to a remote space in order to provide fail safety.
ack(event)

The advantage of using a JTA manager is that you get a single commit for the state replication and your own game specific transaction (database calls would be the top candidate here). However, due to the two phase XA transaction implementations being expensive, JTA will incur a slight performance penalty.

The JTA manager implementation used at the time of writing is Bitronix.

User Transaction

When not using a JTA manager you will most likely start your own user transactions for database calls (and any other transactional resources). This will result in a nested transaction for the event: the *user transaction* (called “game” transaction in the diagram below) will be nested within the *internal* Firebase transaction.



As these two transactions are not connected, it is important to realize that the user transaction can very well be committed without the internal transaction being committed properly. Also a rollback of the internal transaction will not cause a rollback of the user transaction. In this scenario it is therefore important to write robust handling for rollback scenarios for the user transaction.

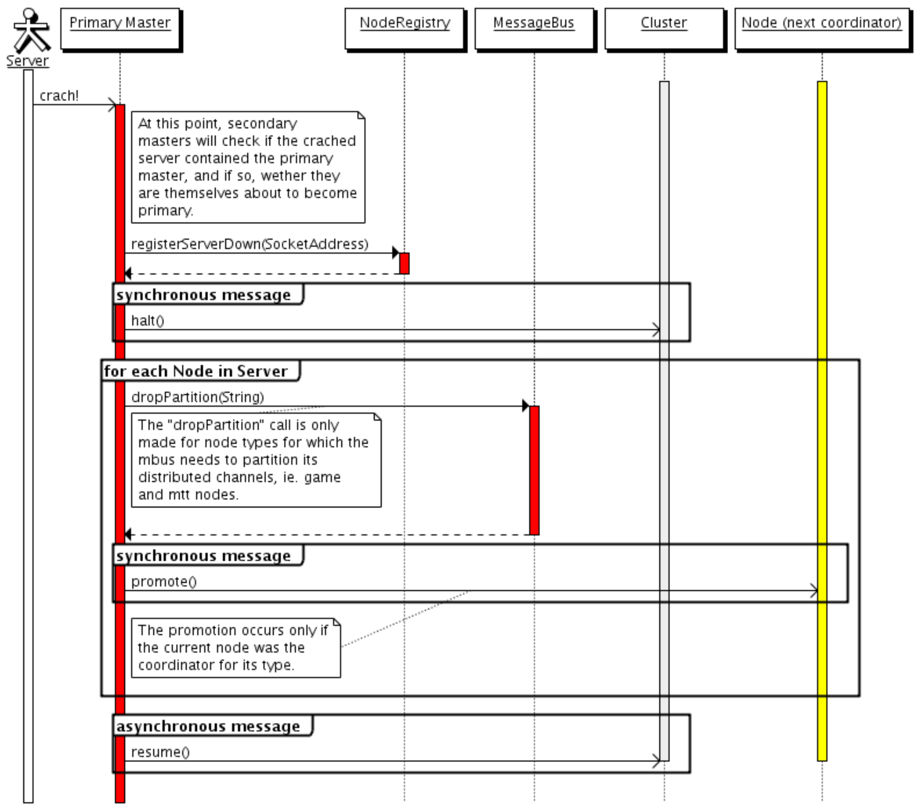
If the user transaction fails and is rolled back, you can throw a `RuntimeException` to roll back the internal transaction. This cannot, be applied the other way around, in other words: a failing internal transaction can never rollback a user transaction.

Failover

When a server experiences a hard, non-recovering crash such as power failure or `` `kill -9` `` on the process, the cluster will detect the loss of a member and assign another server with the same node type to take over the tables or tournaments the lost node was responsible for.

For example, if we have four servers, two servers with both a client and a master node, and two servers with one game node each, then the following happens:

1. All nodes recognize that a node disappeared and immediately halts all execution.
2. The primary master node takes the tables or tournaments that was being executed on the lost node, and distribute them onto the remaining nodes of the same type.
3. When the topography of the cluster have been updated successfully, the primary master sends a “resume” command to all nodes and game play continues.


Server
Primary Master
NodeRegistry
MessageBus
Cluster
Node (next coordinator)
crach!
At this point, secondary masters will check if the crached server contained the primary master, and if so, wether they are themselves about to become primary.
registerServerDown(SocketAddress)
synchronous message
halt()
for each Node in Server
dropPartition(String)
The "dropPartition" call is only made for node types for which the mbus needs to partition its distributed channels, ie. game and mtt nodes.
synchronous message
promote()
The promotion occurs only if the current node was the coordinator for its type.
asynchronous message
resume()

The most complex behaviour when a failover occurs is when the crash will terminate a thread within event execution. As stated in the failing user transaction section, if you have already committed your own user transaction then they are committed and will not be rolled back or cancelled.

The state will be replicated when the internal transaction is committed. The committed state is then what the other game node will read from the space. Since event handling is sequential on a game object we will not lose state for events that have executed fully and that has been committed to the space.

To reiterate, when a game or tournament object has been `put` to the space and the transaction has committed then the state is replicated in the system and will not be lost if the executing server goes down.

However, internal system messaging is not transactional for *outgoing* messages to clients, which can be lost when a server goes down. The only way it would be possible to ensure message delivery would be to engage the message bus in a two-phase commit as well, but this would effectively kill all and any performance. To make a robust failover solution this also needs to be addressed in the application, for example via idempotent messaging, see below.

Failure Scenarios

Failing User Transaction

Let's say you are executing a user event regarding a monetary transaction from an account to the given table. You make the database changes within a *user transaction* and commits, then something goes awry and a runtime exception is thrown after the user transaction is committed. The runtime exception will result in the *internal transaction* getting rolled back. The state of the table have now been rolled back and will not reflect the changes you performed while executing the event prior to the exception, but the database *will* have moved money according to the event.

To properly handle this case you need a robust idempotent handling for the monetary transactions (this is something that is recommended regardless application) so that the money is not moved again, should the client resend the event.

A similar real world use case is where you have a remote server that publishes a service for executing monetary transactions. Calls to such services are normally not contained in a distributed transaction, so if the call never returns you will not know if the call was handled or not.

The same scenario applies for our example. A good way of solving the unknown variable is letting the service manage duplicate calls but do so in an idempotent behaviour. Failing to do so in a game or service implementation will for most systems lead to unwanted behaviour.

Dropped Outgoing Message

As messages sent from a game to connected clients are not included in any of the game transactions, it is possible, even if the probability is often low, to drop outgoing events. For example:

1. Player acts, resulting in event X sent to game.
2. Game processes X and sends answer Y to client
3. Player re-connects while Y is in transit, causing Y to be dropped

The danger here is of course that the game has processed the event, but the player will not realize it and may retry the action that caused the event in the first place. If this is an action which involves a monetary transaction, we need to make sure the game does not allow a player to send the same action several times.

Just as in the failing user transaction, an idempotent behaviour will help. If each unique action carries an equally unique ID, the game can check if the action has already been performed and send back the same answer to the player without actually performing it again. The key here is to protect the game from repeated actions, and also find a way of transparently notifying the player of a result of a previous action.

Persistence

Firebase supports JDBC datasources as deployed units. It also supports JTA for unifying transactions. JPA is supported but not as a separate deployment unit.

JDBC

Data sources can be deployed as XML files in a Firebase installation. They are made available to the game and tournament implementations via a system service.

The current data source pool is c3p0.

The data sources can be accessed either via the data source service, or via JNDI.

JPA

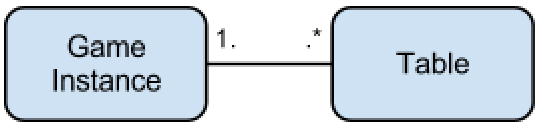
JPA is possible to use in Firebase, but from version 1.9 they are no longer available as separate deployment units. Rather you will package your entities as you would normally and deploy together with the other deployment units.

Platform Domain Data

A game object in Firebase handles only events for tables, almost everything else is managed by the platform. This domain includes lobby management and chat channels. The lobby manages all tables available in the system, and the chat channels are available for the developers to use as a part of the platform.

Tables

A “table” in Firebase is a representation of an area where players join to play a particular game. Tables carries references to the game state and to metadata about the game. They are created by a game activator which controls the number of available tables in the system.

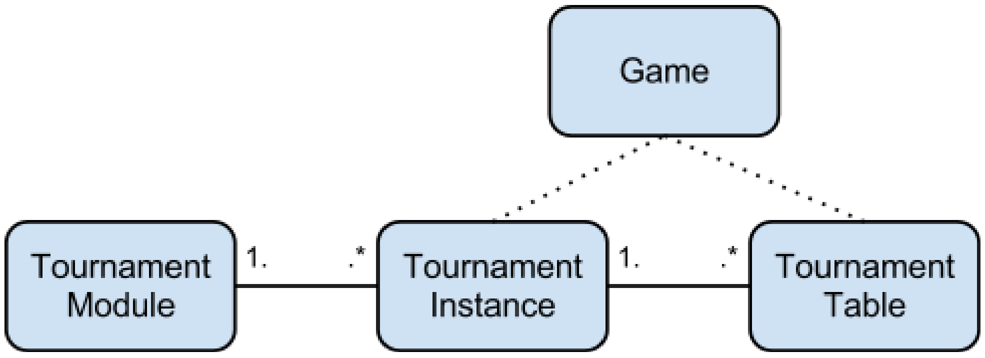


A game instance should never keep references to tables as the execution of events for a given table may be moved between servers in a Firebase cluster.

Tournaments

Tournament instances are created by a tournament activator, and similarly to tables, they are representation of an area where players join to play a game, but one that may involve multiple tables.

A tournament instance is associated with a particular game, but the tournament module itself, and hence the activator is not necessarily coupled to a specific game: it is quite possible to write a tournament module that is game agnostic and which can be used across several different games.



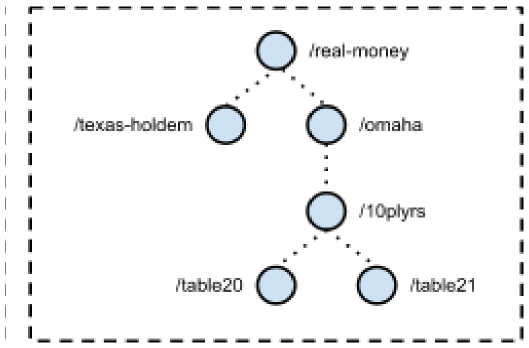
A tournament table is an ordinary table, but one that is created by a tournament instance. Execution of events for a given tournament table is still taking place on game nodes, whereas events for a given tournament instance is executing at a tournament node.

Lobby Data

All tables and tournaments created in the system are automatically arranged in a lobby. The lobby is a data structure organising its members in a tree of objects. Each object in the lobby has a path, and a set of attributes describing the table or tournament. For example:

```
/real-money/omaha/10plyrs/table21[speed=fast,seats=10,id=21]
```

Together with more tables a hierarchy is composed of branches with the objects as leaves. Tables or tournaments are always the leaves, and only tables or tournaments carries attributes.



The lobby topography is created by the activators when tables and tournaments are created, by specifying a path for the new objects.

The structure of the lobby as static in the sense that objects within a lobby can never change places, and the structure is defined when a game is developed. Client and server developers will agree on a lobby model to use, the server game developers populate the lobby when the tables and tournaments are created. The client developers uses a subscription model with delta changes to manage the lobby representation in the actual game clients.

Subscription and Delta Changes

Connected clients use a subscription model with delta changes to manage the lobby representation in the actual game clients. There are two major concerns addressed with this model:

- Partition of data: The hierarchy partitions data naturally, and a client can subscribe to a subset of the entire tree in order to cut down on data bandwidth and processing.
- Bandwidth: The above partitioning together with delta changes makes this model very efficient. This is important for device programming but also for installations where bandwidth is expensive.

Clients “subscribe” and “unsubscribe” to subsets of the lobby. Upon a subscription request Firebase will first respond with a complete set of node data for the subscribed area, after which changes in the same will propagate as delta change events to subscribers.

For efficiency reasons delta changes are accumulated into batches and pushed to the subscribers in intervals, the default of which is two seconds.

Chat Channels

The chat channels managed by Firebase work much like ordinary IRC channels. If a channel does not exist it will be created when the first message is sent and client can choose to listen or send chat events for particular channels. Developers can also attach filters, via the `ChatFilter` service, in a Firebase server to monitor or filter existing channels.

Users and Clients

This section very briefly discusses user and client scenarios as they appear in Firebase. For simplicity, a user in this section is equalled to a connected client.

The normal use case for a user is:

- “connect” - Opening a communication channel to the server.
- “login” - Authenticating user with username and password.
- “join/play/leave” - Join tables, play games and leave tables.
- “logout” - End client session and close communication channel.

Connect

Client connects to Firebase by establishing a connection with one of the transports. Port and path locations are configured by server, with the following defaults:

- 0.0.0.0:4123 - Styx binary transport
- 0.0.0.0:8080 - “/socket” - Web socket HTTP transport
- 0.0.0.0:8080 - “/cometd” - Comet HTTP transport

Must communication with a Firebase server requires the client to authenticate with a login packet, with the following exceptions:

- Authentication can be performed with a login packet.
- The lobby can be queried and subscribed to.
- The server version can be queried as well as specific game versions

It is highly recommended that the client verifies the platform protocol version first and then verifies the specific game version before continuing. If a mismatch is detected it is usually not be safe to continue.

Login

A client can send a login request with a login packet. The packet contains fields for username, password, operator ID and a free byte field for more complex credentials. When received Firebase will try to locate a `LoginHandler` to manage the authentication.

If a client logs in with a player ID that is already logged in, the first client will be forcibly logged out and its socket will be closed.

If a client was disconnected while sitting at tables, and reconnects before he has timed out, the tables will be notified when he returns. Also the client itself will get a list of tables and tournaments he is currently participating in.

Join / Play / Leave

When a client wishes to join a game he uses a “join table” request. This packet is handled by Firebase, and the game node will interact with the game in order to allow or deny the join request.

Game play is handled by the game processor. Actions sent between the game and the client for each game is outside the Firebase domain, and Firebase will treat them as byte arrays. It is up to the game developer to formulate a protocol and an encoding for a particular game.

When a player wishes to leave a table he does so with a “leave table” request. Again, this packet is handled by Firebase in cooperation with the game.

Logout

Then the server receives a logout request it will clean up resources for the client on the server:

- Unwatch on all tables client is watching.
- Leave all tables client is seated at, if requested by the logout command.
- Remove client from local client registry.
- Remove client from distributed client registry.

Strictly speaking this will also occur on disconnects, but client developers should strive to send the logout packet to enable a cleaner execution.

Disconnects

If a user disconnects - loses connection without a logout - the system will take the following actions:

- Unsubscribe client from all lobby subscriptions.
- Update player status to “waiting rejoin” in system state.
- Update player status to “waiting rejoin” on all tables the player was joined at.
- Unwatch all relevant tables.
- Remove client from local registry.

When the client has been in status “waiting rejoin” in the system state for a set timeout period a reaper subsystem in Firebase will remove the client from the system state and cleanup the player resources similarly to a logout.

If the client reconnects before the timeout, he will receive notifications for each table watching and seated at. However, lobby subscriptions and filtered join requests will not be resumed on reconnections, so truly transparent fail-overs need to re-subscribe to any lobby data.

If the client reconnects after the reaper has removed the client, then the reference to seated tables etc. have been cleaned up and it will not receive any notifications even if the game still has him seated at a table, see more below.

Connection Status and Tables

Firebase uses login, logout, join, leave and similar commands together with the actual transport status to determine if a client status. However, this status may be different from the view of a player at a table. To see why, consider the following poker scenario:

1. A player is sits at the table
2. Hi disconnects immediately after he placed a buy-in
3. The reaper timeout the client for the player *before* the hand is ended

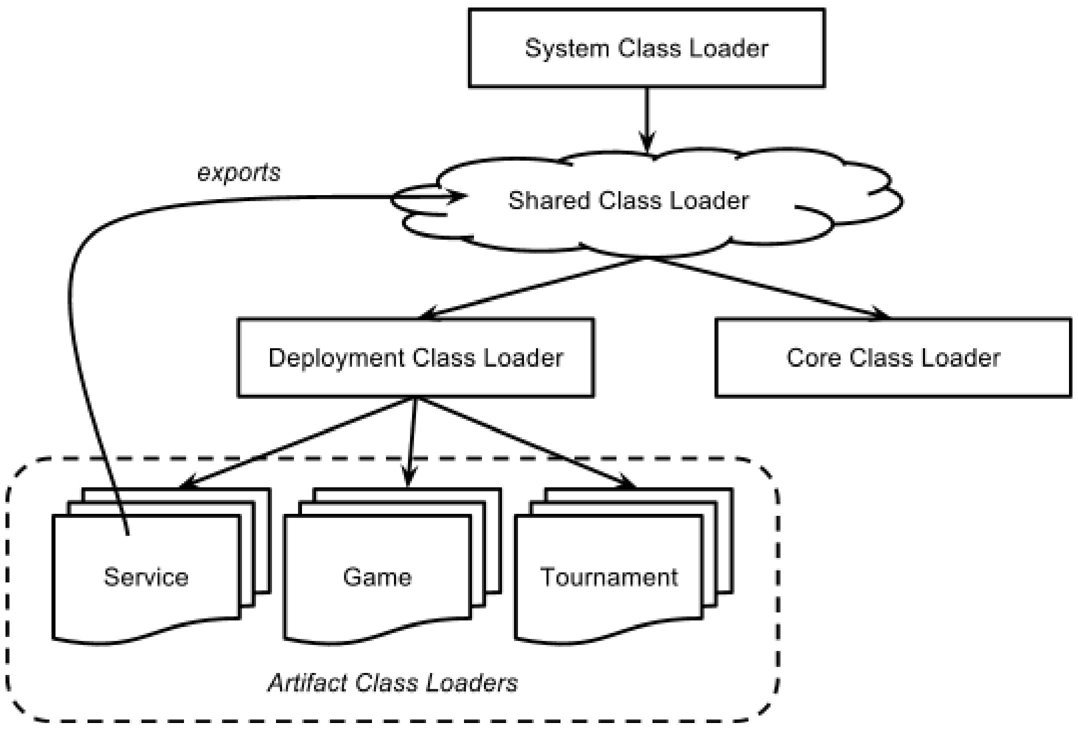
As we can see the player *cannot* be removed from the table even if the reaper has collected him. From this follows that it is possible for a player to be sitting in a game, even if the reaper has removed his client, and the game developer must not remove the player indiscriminately when he disconnects.

Class Loading

Firebase keeps a class loading hierarchy in place to support code isolation and make hot deployment of code units possible. Each game, service and tournament will be loaded in its own private class loader. Games and tournaments communicate only by the defined game API, whereas Services can export classes, in effect making them usable in the entire server.

Hierarchy

Class loaders are formed in a hierarchy use parent / child relationships with parent first class loading priority. The “Shared Class Loader” is a parent of all internal class loaders and is used to expose Service interfaces and dependencies via the method called exporting.



The above class loaders are sourced from different locations in the Firebase installation directory, with the exception if the 'shared' class loader which maintains an export table for classes that need to be available through the entire server.

- The system class loader is created with the Java VM. Normally this contains all JRE classes as well as those located in `lib/common` in a Firebase installation.
- The shared class loader maintains all public service classes and is not source from any disc location. See exporting below.
- The core class loader loads all internal Firebase classes from `lib/internal`. These are not visible to deployed artifacts.
- The deployment class loader is a shared parent of all deployed artifacts. It is source from the `game/lib` folder and all loaded classes are visible to deployed artifacts.
- The artifact class loaders are responsible for a single deployed unit each, ie games, services or tournaments. Artifacts are only visible to the local class loader and are not shared between deployment units.

Exporting

Services contracts are deployed and made available across the entire server. This is done via the method called "exporting", where the service contract is automatically exported to the shared class loader by virtue of extending the Contract interface.

Class referenced by the service contract must either also be exported, by explicitly declaring them in the service descriptor, or be placed in `lib/common` to be loaded by the system class loader. This is because service contracts are loaded by the shared class loader in order to be accessible through the entire server.

- The Contract interface itself of a service is always exported by Firebase.
- *Any* classes or dependency the Contract have must be explicitly exported by the developer.

Exported classes and packages are declared in the service descriptor.

Shared Libraries

A common task will be to share utility libraries across games and services. Understanding of the class loader hierarchy is vital to achieve this, but in short the following short rules apply:

- To share libraries across games or services, place them in `game/lib`.
- To share libraries across services, either have a service export them and make the other services dependent on the exporting service, or place them in `lib/common`.

If both games and services needs to share classes they can only be placed in `game/lib` or `lib/common`. However, it is worth noticing that in the system class loader will never be eligible for hot redeployment, and will be universal for the entire server.

HTML5

Firebase comes with HTML5 support enabled by default, in two flavors: Web Socket and CometD. Neither of them requires any changes in the actual game code, only the client and the protocol are changed.

Protocol / JSON

The Firebase protocol is described in a a speparate document. It is worth noting that the procol object format does not change, but is only encoded as JSON instead of the binary Styx packaging. So for example, a login request from an HTML5 client would look like this:

```
{
  "user":"dummyUser",
  "password":"666",
  "operatorid":0,
  "credentials":"",
  "classId":10
}
```

The only difference between the binary Styx binary encoding and JSON is that in JSON the field names are always present and that the "classId" is included, which corresponds to the type ID:s in the specification.

Mount Points

By default Firebase enables both Web Socket and CometD support mounted at the following contexts on localhost port 8080:

- `/socket` - This is the web socket mount point.
- `/cometd/*` - This is the mount point for CometD.

By default Firebase also enables serving of static content from the directory web/game in an installation directory. This content will be mounted under /static/* in the web server.

JNDI

From version 1.9 Firebase supports JNDI for accessing JDBC data sources and the system transaction manager. The JNDI context is mounted for runtime execution, and is a read-only context within the “java:” namespace. The following objects are available:

- java:comp/env/TransactionManager - The JTA transaction manager
- java:comp/env/UserTransaction - The JTA user transaction
- java:comp/env/jdbc/<data-source-name> - One entry per data source

Server Development

This part of the reference manual deals with the server side development if games, services and tournaments. It also outline available services in Firebase, building and best practises.

Server Game Development

What's a Game?

A *game* in Firebase is a class which implements the Game interface. It interacts with incoming actions around a *table* with *players*. A game is packaged as a deployable unit and can be executed on a Firebase server.

Firebase encourages the separation of logic and state for the games. A game's logic should in no way be entangled with the state, but rather be able to take an event and state and figure out what to do and evolve the state further.

The game logic is responsible for executing events on tables, and the tables are the holders for the game state. For each event sent by a client, the logic will be called with the event itself and the table the event was sent to.

Table and State

A game is associated with a number of *tables*. A table is an *area* of game play. Players can join, leave and participate at a game at a table. The table also holds the game state in memory; this is useful for volatile game state that does not have to be written to a database.

Game Life Cycle

A game has a life cycle which is managed by the platform. All games implements the `Initializable` interface which specifies two control methods used by Firebase to start and stop a game instance:

- `init(GameContext)` - Initialise the game for a given context. This method is called once by Firebase, and the game may throw a SystemException to indicate that the initialisation failed.
- `destroy()` - This method is called when the game is removed from a server. The game should use it to clean up resources.

The Game Interfaces

A game instance is composed of one or more interface implementations. By implementing different interfaces the game can participate in different ways with the Firebase server. Of these interfaces only Game is mandatory.

Game

The Game interface is mandatory. It specifies two life time methods for the game instance, and one accessor for the processor responsible for the game. Below is an abbreviated view of the game interface, please refer to the Java API documentation for more information.

```
public interface Game [...] {

  [...]

  /**
   * Get the Game Data processor class for actions. This method will
   * be accessed concurrently, but only for one table at a time. By
   * returning new game processors for each call, the game processors
   * does not need to be concurrent.
   *
   * @return A game processor, never null
   */
  public GameProcessor getGameProcessor();

}
```

For each event at a table Firebase well ask the game for a processor. The game processor can be shared across events, but then the developer need to worry about concurrent execution. Therefore most implementation follows something like this pattern:

```
public class MyGame implements Game {

  public void init(GameContext c) { }

  public GameProcessor getGameProcessor() {
    return new Processor();
  }

  public void destroy() { }

}
```

The “Processor” returned above does not need to be synchronized unless it share resources with other processors. Firebase guarantees that execution for a particular table is done single threaded, so by recreating the processor concurrency becomes a non-issue.

Game Processor

The game processor is the key interface for a game. It is where a game implementation handles logic for an incoming action, for a particular table.

> **Note:** A processor handles *one* action for *one* table at the time. Firebase guarantees that this process will be single threaded.

The processor is created by the game, and is usually created per invocation. It contains two methods for handling incoming actions:

```
public interface GameProcessor {

  public void handle(GameDataAction action, Table table);

  public void handle(GameObjectAction action, Table table);

}
```

Each invocation have an action and a table. Firebase guarantees that execution for a particular table is done one single threaded, and the context for the action is isolated for an invocation on the game processor. Firebase differentiates between *data* actions and *object* actions.

Game *data* actions are what is send to and from connected clients. The data in question is a byte array: Firebase does not mandate what kind of protocol you use, and as such the data is transported back and forward as bytes.

Game *object* actions are either actions sent by services or other components within Firebase, or actions that have been scheduled for later execution. A scheduled action is, as opposed to a *data* action a POJO as it will never leave the environment of the Firebase server.

There is a transaction wrapping each execution of an event in Firebase. This means the game state, any outgoing actions, and optionally attached JTA data sources, are pending the successful execution of the event. The table can rollback this transaction by throwing a runtime exception.

Table Interceptor

The table interceptor interface can be implemented in order to control join, leave and reservation requests on the table, which are otherwise handled transparently by Firebase.

For example, if a player is involved in a game, that for some reason does not allow players to leave mid-game, the table interceptor can return false when the `allowLeave(Table, int)` method is called for a player who is currently in the game. It is up to the game to then remember that the player wants to leave and then remove the player at a later time, if that is the desirable behaviour.

The implementation should be made available to Firebase via a “provider method”, and is indicated by implementing the corresponding provider interface:

```
public class MyGame implements Game, TableInterceptorProvider {

  public TableInterceptor getTableProvider(Table table) {
    return new Interceptor();
  }

  [...]

}
```

Firebase will recognize the provider interface above and use the table interceptor to cooperate around join and leave requests. Again, Firebase guarantees that invocations around a single table are single threaded so by creating new interceptors for each invocation the game becomes thread safe.

Table Listener

The TableListener interface can be implemented to listen for table events that are handled external to the game by Firebase. These events include join, leave, reservations, status changes etc. A common use is to listen for `playerJoined(Table, GenericPlayer)` to send initial game data when a new player have joined the game:

```
public class Listener implements TableListener {

  @Override
  public void playerJoined(Table table, GenericPlayer player) {
    int playerId = player.getPlayerId();
    String msg = "Hello " + player.getName() + "!";
    GameDataAction action = createAction(table.getId(), playerId, msg);
    GameNotifier notifier = table.getNotifier();
    notifier.notifyPlayer(playerId, action);
  }

  private GameDataAction createAction(int tableId, int playerId, String msg) {
    GameDataAction action = new GameDataAction(playerId, tableId);
    ByteBuffer bytes = ByteBuffer.wrap(msg.getBytes());
    action.setData(bytes);
    return action;
  }

  [...]
}
```

Just as in the case with the table interceptor, the game implements a provider interface to signal to Firebase that it wants to listen for tabel events. Firebase will recognize the provider interface and signal to the provided listener on events.

```
public class MyGame implements Game, TableListenerProvider {

  public TableListener getTableListener(Table table) {
    return new Listener();
  }

  [...]

}
```

Firebase guarantees that invocations around a single table are single threaded so by creating new listeners for each invocation the game becomes thread safe.

Tournament Game

If a game participates in a tournament, the `TournamentGame` interface must be implemented. It provides a tournament processor used to start and stop tournament *rounds*. Please refer to the tournament section for more details.

Sending Events

A `GameNotifier` object is provided to the games for purposes of sending events to clients connected to the Firebase server and associated with a table. The game notifier is provided by the the table and is unique for each invocation.

> **Note:** The notifier is unique within the context of an executing action. When the action is finished processing the notifier instance is dropped. Therefore game implementations must not keep references to notifiers.

The game notifier has a number of methods for notifying players of events. The different use cases are typically events that should:

- Reach all players, including watchers. For example, "player A did did X".
- Reach all players, but not watchers. For example, "click 'yes' to start the game now".
- Reach only one player, for "secret" data, such as a player's cards.
- Reach all players, except for one player. This is used in the case where the player who acted gets a hidden response, and the other players should just know that he did something.

All methods take a single, or a list of, `GameEvents`, which will usually be of the type `GameDataAction`. As Firebase is game agnostic, it doesn’t impose any particular protocol between the server game and connected clients. And as such all game data actions carries its payload as bytes only.

Below is a simple example of sending a string action to all players seated at a table:

```
public class MyProcessor implements GameProcessor {

  [...]

  private void sendHello(Table table) throws IOException {
    GameNotifier notifier = table.getNotifier();
    // Get UTF-8 bytes and wrap
    byte[] bytes = "Hellow World".getBytes("UTF-8");
    ByteBuffer buff = ByteBuffer.wrap(bytes);
    // Create action, we'll use -1 as player ID
    // as this action comes from the server and
    // not any individual player
    GameDataAction data = new GameDataAction(-1, table.getId());
    data.setData(buff);
    // Send to all joined players, but not
    // to the watching players
    notifier.notifyAllPlayers(data, false);
  }
}
```

Normally a game will have its own protocol objects and transport encoding. The wrapping of an internal event will then be handled away to utility methods.

```
public static GameDataAction toByteAction(MyAction act) {
  int playerId = // current player, or -1 if "from server"
  int tableId = // current table, eg: table.getId();
  GameDataAction dataAction = new GameDataAction(playerId, tableId);
  ByteBuffer action = act.toBytes(); // byte encoding of action
  dataAction.setData(action);
  return dataAction;
}
```

Sending Events Asynchronously

While sending actions to players are recommended to be done when executing an action on the Table, Firebase does support sending events in an asynchronous manner. An example of this could be if you need to fork a separate thread to perform something that takes a long time and want to notify the player when done. Note that events sent this way will not be part of the transaction that takes place when an event executes on a Table; i.e. if you trigger a rollback on a Table then actions send asynchronous to clients will not be rolled back.

To send events asynchronously you can either create your own Service that implements `RoutableService` or you can access the public service called `RouterService` which exposes methods for routing internal events.

Transaction

As interactions with the notifier takes place in the context of an executing event, via the game processor, there is a wrapping transaction involved. This means no actions will be sent until the game processor “handle” method has finished executing without errors. Should the game throw a runtime exception after having sent actions via the notifier, but prior to the end of “handle”, such action *will not* be sent to the intended clients.

Scheduling Events

Firebase support loopback scheduling by providing a table scheduler via the table instance `getScheduler()` method. Scheduled game actions are delivered back to the game after a given delay in milliseconds.

As Firebase may be run on a cluster and support failover and high availability, game implementation should prefer the table scheduler over other timers, because then...

- … you’re guaranteed one action at the time per table.
- … scheduling will work in a failover scenario.

To schedule events for later execution you use the table scheduler from the table:

```
TableScheduler sch = table.getScheduler();
// schedule action with 1000 millis delay (1 sec)
UUID id = sch.scheduleAction(dataAction, 1000);
```

The most common thing to schedule is to check if a player has acted within a specific time. For example, a poker player will normally have 20-30 seconds to act before he is auto-folded and game play continues to the next player. So the poker server will handle an action from a player, move to the next player and schedule a timeout.

Scheduled actions can be cancelled. Each scheduled action is associated with a UUID which can be saved for later use. Given the above example, if the player acts in time it is prudent to remove the timeout when he does:

```
TableScheduler sch = table.getScheduler();
MyPlayer player = myGameState.getCurrentPlayer();
UUID id = player.getTimeoutActionId();
if(id != null) {
    sch.cancelScheduledAction(id);
}
```

Transaction

Scheduled events are also attached to the current event transaction. So if a game schedules an event, and then later throws a runtime exception, Firebase will rollback its internal transaction and the scheduled event will be cancelled.

Game State

The table instance contains a reference to a “game state”. This is a plain old Java object which can be used by the game to store volatile information which will survive the execution of the current event. Typically a game will store all of its state in the table, and supplement this with writing significant event to a data store.

```
MyStateObject state = (MyStateObject) table.getGameState().getState();
```

The state can be set in the game, but more commonly it is created by the activator when a table is initially created. A state object can be manipulated during event execution but does not need to be “set” on the table again, changes will be tracked irregardless.

The game state is available via the table, and it participates in the event transaction, and also in the Firebase Enterprise Edition clustering. This means that if the game throws a runtime exception during execution of an event, no changes to the game state will be propagated in the system as the transaction is rolled back.

The game state must be serializable in order to support a full cluster deployment. For high performance systems, the size of the game state may be crucial as well. You may not realize you have serialization problems when you develop though: Firebase will optimize the execution and only partly serialize your object. To check if the deserialization works during development, start Firebase with this flag:

```
-Dcom.cubeia.forceDeserialization
```

If a game transaction commit successfully the game state will be propagated over a Firebase cluster. Firebase uses a so called buddy replication which maintains at least two copies of a particular state in a cluster to enable failover.

Game Activation

When a Firebase server starts up there are no tables created in the system. Responsible for this is a *game activator* and the process is called *game activation.* The game activator will be initiated by Firebase and has the means to create and destroy tables. In a Firebase cluster the activator will be created on the coordinator game node only.

The Game Activator

Firebase contains a default activator which kicks in by default. A game can choose to configure and use its own activator via the game descriptor. For example:

```
<game-definition id="112">
  <name>MyGame</name>
  <version>v0.2</version>
  <classname>com.mygame.server.MyGame</classname>
  <activator>com.mygame.server.MyGameActivator</activator>
</game-definition>
```

The configured “activator” element must indicate a class which implements the interface `GameActivator` and has a default constructor.

If no activator is configured the system `DefaultActivator` will be used. The activator is primarily responsible for creating and destroying tables. It is guaranteed to be a singleton within a cluster, ie. if Firebase is configured as a cluster with many game nodes there will only ever be one activator created at any time.

Lifetime

Firebase will only ever create one activator at any given time. The activator has the following lifetime methods:

- “init” - This method is called when the activator is first created. Normally this occurs when a Firebase installation starts for the first time. However, it is possible for an activator to be created and initialized at a later time if the server where the activator was first created crashes or have been stopped. As such this method will probably do something like this, in pseudo-code:

```
if(!haveTables(activatorContext)) {
  // no tables exist, this is a startup
  // so create initial tables
  createInitialTables(activatorContext);
}
```

- “start” - At this point Firebase is about to start execution. If the game does not have a fixed number of tables it should use an internal thread to control the number of tables at a given time. For example, in pseudo-code:

```
executor = Executors.newSingleThreadScheduledExecutor();
executor.scheduleWithFixedDelay(
    new MyTableCheckTask(),
    10,
    10,
    TimeUnit.SECONDS);
}
```

- “stop” - Firebase is shutting down. If there is an executor running, as in step 2 above, it should be stopped:

```
executor.shutdownNow();
```

- “destroy” - Final call. Please clean up any open resources.

Helper Objects

The `ActivatorContext` contains a number of objects that are used to query and command the Firebase context. These includes:

- `ConfigSource` - This is the configuration for the activator if one exists.
- `TableFactory` - This object is used to create and destroy table in the system. The creation of tables is done by Firebase and the activator will supply yet another helper object called CreationParticipant when creating that table. This second helper object may specify the position in the lobby, its name etc. The table factory is also responsible for listing existing tables.
- `ServiceRegistry` - Service registry accessor.

Configuration

The activator can be configured externally to its archive. To do this a file is deployed, ie. copied into the server deployment folder, with the file name ending “-ga.xml”. This file is matched with the GAR file name, so that if you remove the GAR extension and add "-ga.xml" you will get the activator configuration file. For example:

```
./game/deploy/myTestGame.gar
./game/deploy/myTestGame.ga-xml
```

The configuration will be available as a ConfigSource via the ActivatorContext. Even though the file extension must be XML, the file itself may actually contain any data as it is treated as bytes by Firebase.

The activator may add a configuration listener to its context. In such case it will be notified when the configuration changes, this give the activator an opportunity to change parameters during runtime via its configuration file.

The Default Activator

The default activator is a simple instance of a `GameActivator` shipped with Firebase in the API classes. It uses a very simple naming pattern and lobby path. It can be configured via an XML file in the following format:

```
<activator>
  <!--
    - Frequency in wich the activator scans the available tables
    - to check for changes in millis. Default value is 5 seconds.
    -->
  <scan-frequency>5000</scan-frequency>
  <initial-delay>10000</initial-delay>
  <tables>
    <!-- Number of seats at the table -->
    <seats>10</seats>
    <!-- Minimum number of tables -->
    <min-tables>10</min-tables>
    <!-- Min available empty tables -->
    <min-available-tables>10</min-available-tables>
    <!-- Increment size, ie. how many
       - tables do we create at one time -->
    <increment-size>10</increment-size>
    <!-- Timeout value for empty tables in millis.
       - Default value is 2 minutes. -->
    <timeout>120000</timeout>
    </tables>
</activator>
```

The `DefaultActivator` may be sub-classed, but developers are encouraged to write their own activator instead as the `DefaultActivator` is primarily intended as an example.

Packaging

A game is packaged in a Game Archive, or GAR file. The game archive is defined as a ZIP file with the extension "gar" with the following folder structure:

```
/[*.jar]
/GAME-INF/game.xml
/GAME-INF/[lib/*.jar]
```

FIrebase will scan the above folders for JAR files to add to the classpath before initiating a game. Recommended practice puts the game implementation in a JAR file in the root of the GAR and any dependencies in `GAME-INF/lib`.

Game Descriptor

The game descriptor is mandatory in a game archive. It describes the game and points out the game class and an optional game activator:

```
<game-definition id="112">
  <name>MyGame</name>
  <version>v0.2</version>
  <classname>com.mygame.server.MyGame</classname>
  <activator>com.mygame.server.MyGameActivator</activator>
</game-definition>
```

The above example specifies that `MyGame` is the class implementing the `Game` interface and the `MyGameActivator` implements the `GameActivator` interface. The following elements are used:

- “game-definition” - Mandatory root element. Must contain an integer ID attribute which must be unique per game in a Firebase installation.
    - “name” - Mandatory. Simple game name, used for logging and monitoring.
    - “version” - Mandatory. Simple game version, used for logging and monitoring.
    - “classname” - Mandatory. The fully qualified name of the class implementing the game interface.
    - “activator” - Optional. The fully qualified class name of a class implementing the game activator interface.

The game descriptor is placed in a `META-INF` folder in the game archive with the name “game.xml” and is mandatory.

Deployment

When a game is packaged, it can be *deployed* in a Firebase server. This is simply the process of copying the GAR file to the `game/deploy` folder of a Firebase installation.

Currently Firebase needs to restarted for a game to be picked up or changes to be loaded. Hot deploy and re-deploy is on the roadmap. A Firebase cluster can be updated using rolling restarts where one server is restarted at the time, but single servers needs to be restarted completely for new deployments to be picked up.

Tournament Development

What is a Tournament?

A *tournament* is an organised competition in which many participants play each other on individual tables. After each game, or *round*, one or more participant is either dropped from the tournament, or advances to play a new opponent in the next round. Usually, all the rounds of the tournament lead up to the “finals”, in which the only remaining participants play, and the winner of the finals is the winner of the entire tournament.

A tournament is distributed over one or more tables and are commonly referred to as Multi Table Tournaments (MTT). A tournament that is played out on a single table is commonly referred to as a Single Table Tournament (STT).

Firebase contains an API for writing tournaments which makes it possible to separate tournament logic from game logic. Tournaments are also deployed separately from the games.

A tournament class implements the `MTTLogic` interface. Much like games are separated from their states, which are encapsulated by the table instances, the tournament logic is separated from particular tournament states, these being encapsulated in `MttInstance` objects. The tournament instance acts for the tournament logic exactly like the table to the game processor: the tournament logic will get a callback for every action in a tournament, together with the instance it occurred at.

Logic and Instances

The tournament *logic* will be instantiated only once in a server and should not share any state between calls to `handle(MttAction, MttInstance)`. This method corresponds to the game processor event handling.

The tournament *instance* is not directly controlled by the tournament itself, but passed into the tournament logic each time an action needs to be handled. Firebase guarantees that each call to "handle" on tournament logic is done single threaded *per tournament instance*, so that two actions will never be executed at the same time for the same tournament instance.

Tournament and Games

The actual game play of a tournament will take place at tables and as such, the rules of a particular game will be used. The tournament will manage *only* the tournament specific events, the game events will be handled by the tables and games.

For a game to participate in a tournament it needs to implement the tournament game interface. This is because a tournament may need to control "rounds" across multiple tables. There is also a tournament table listener that the game can use to handle when tournament players join and leave the table.

Tournament Life Cycle

A tournament logic object has a lifecycle which is managed by the platform. It is important not to make assumptions about a particular tournament logic: the platform may start/stop objects at any time. It may keep several instances alive at the same time and it may create the tournaments lazily.

However, firebase will create at least one instance of the logic that will handle the incoming actions regarding a specific tournament instance.

A typical tournament instance lifecycle looks like this:

- A tournament instance is created by the tournament activator. The tournament appears in the lobby.
- The tournament logic schedules common state changes such as "announced", "registering", "start", "running" and "finished". These usually corresponds to lobby attributes so that connected clients now which state the tournament is in.

- A player interacts with the tournament by sending `MttAction` events to the tournament. These are handled by the tournament logic.
- When a tournament starts, the tournament logic uses a “table creator” factory to create tables for the tournament, it then seats the registered player at the tables.
- The tournament logic start the game play by sending a `StartMttRoundAction` to all tables.
- When a “round” is finished at a table, the game reports the results to the tournament by sending an `MttRoundReportAction` to the tournament. It then waits for the tournament to decide what to do next.
- The tournament logic kick players, reseat players, create and destroy tables and otherwise manage the play between rounds.
- When the tournament is “finished” the tournament activator picks it up and destroys the instance. The tournament disappears from the lobby.

Tournament Interfaces

A tournament is composed of a set of cooperating interfaces the receives and sends events to players and each others.

MTT Logic

This is the main interface for a particular *type* of tournament. You configure Firebase to use this class in the tournament descriptor. The tournament logic will get methods invoked by Firebase when the following happens:

- A tournament is created - When the activator creates a tournament Firebase calls this method in order for the tournament logic to react to the creation. Normally the logic will schedule state-changing actions for the instance and setup any prerequisites.
- On actions - When a player, a game or a service sends an action to a tournament the logic will get its `handle` method invoked. Firebase guarantees that actions for the same *instance* will never be executed at the same time.
- A tournament is destroyed - When the activator destroys a tournament Firebase calls this method. Normally the logic will cancel any scheduled actions, and if the tournament is not yet finished, inform any remaining players that it is closing.

A tournament logic is separated from its instances and should not keep any references to them as Firebase may move the execution of a particular instance from one server to another without notice.

Tournament Game

In order for a game to participate in a tournament, it needs to implement the interface `TournamentGame`, which exposes a `TournamentProcessor` to Firebase. This processor is used to start and stop rounds in a tournament, via the following methods:

- `startRound(Table)` - Called when the tournament logic has sent a `StartMttRoundAction` to a given table.
- `stopRound(Table)` - Called when the tournament logic has sent a `StopMttRoundAction` to a given table. This is more unusual, as normally game play at a table stop by itself, for example after a hand, but may be used to interrupt game play.

Tournament Table Listener

Players to not join or leave tables in a tournament. Rather they are seated or unseated by the tournament logic. If the game needs to be aware of this process it should implement the table listener provider interface, and make the returned table listener implement `TournamentTableListener`. Firebase will then notify it when players join, rejoin and is removed by the tournament logic.

Sending Events

The tournament sends events to players or tables via an `MttNotifier` instance which is available in the tournament instance object.

Sending Events Asynchronously

While sending actions to players are recommended to be done when executing an action on the tournament, Firebase does support sending events in an asynchronous manner. To send events asynchronously you can either create your own Service that implements `RoutableService` or you can access the public service called `RouterService` which exposes methods for routing internal events.

Transaction

For all executing events in Firebase there is a surrounding transaction. This means that events sent via the MTT notifier will not be flushed until the execution of the current event is successfully finished. The tournament logic may interrupt the current transaction with a runtime exception in which case no event will be sent from the notifier.

Scheduling Events

In order to control state changes and other timing effects a tournament should schedule events via a scheduler found in the the tournament instance. By doing this the scheduled event will be executed in its own transaction, and also will survive if a failover occurs and Firebase redirects the execution of a particular tournament instance to a new node.

```
public void hand(MttAction act, MttInstance inst) {
  MttAction next = // do something with actions here
  Scheduler<MttAction> sch = inst.getScheduler();
  sch.schedule(next, 1000);
}
```

This scheduler works precisely as the game scheduler, so please review that section for more information.

Tournament Activation

When a Firebase server starts up there are now tournament instances in the system. In order to create instances Firebase uses a *tournament activator* which get the responsibility of creating and destroying instances.

> **Note:** Contrary to the game activator there is no default implementation shipped with Firebase. This means the activator is mandatory in the tournament descriptor.

The Tournament Activator

The tournament activator is instantiated when a tournament coordinator node is started. The tournament developer specifies it in the tournament descriptor, and Firebase will instantiate the class and manage its lifetime.

Lifetime

Just as the game activator the tournament activator has four lifetime methods called by Firebase. An activator will normally start an internal thread in the background to do maintenance outside these methods.

- `init(ActivatorContext)` - This method is called when the activator is first created. It should setup resources and configuration.
- `start()` - At this point the tournament node is starting and the activator should check if it needs to create any new tournament instances, and then schedule a background thread to keep the tournaments updated.
- `stop()` - The tournament node is stopping. The activator should close and destroy all remaining tournaments and also cancel any background thread.
- `destroy()` - Release any remaining resources.

Helper Objects

The tournament activator contains contains a number of helper objects for the tournament to interact with. These includes the tournament factory used to create new tournaments but also a router for sending events.

- `ActivatorRouter` - This object can be used to send events to games and tournaments. If an activator want to receive events it should implement the `RoutableActivator` interface.
- `ConfigSource` - This is a binary source for a deployed configuration.
- `ServiceRegistry` - Access to the system services.

The tournament factory used to create and destroy tournaments is set on the activator instance before `init` is called. This factory also contains methods for listing available tournament instances in the system.

Configuration

The activator can be configured externally. To do this a file is deployed with the file name ending "-ta.xml". This file is matched with the TAR filename, so that if you remove the TAR extension and add "-ta.xml" you will get the activator configuration file. For example:

```
game/deploy/myTestTournament.tar
game/deploy/myTestTournament-ta.xml
```

The configuration will be available as a `ConfigSource` via the `ActivatorContext`. Even though the file extension must be XML, the file itself may actually contain any data as it is treated as a binary file by Firebase.

The activator may add a configuration listener to its context. In such case it will be notified when the configuration changes, this give the activator an opportunity to change parameters during runtime via its configuration file.

Packaging

The tournament logic and its supporting classes are packaged in a Tournament Archive, or TAR file, with a “tar” extension. This is a ZIP file where the contents must match:

```
/[*.jar]
/META-INF/tournament.xml
/META-INF/[lib/*.jar]
```

All JAR files will be picked up and added to the tournament class loader. Normally the tournament module and its classes will be packaged in a JAR file in the archive root and all dependencies in `META-INF/lib`.

Tournament Descriptor

The tournament descriptor is an XML file used to configure Firebase for a particular tournament deployment. For example:

```
<tournament-definition id="998">
    <name>Systest Tournament</name>
    <version>v0.0.1</version>
    <classname>com.cubeia.test.systest.tournament.Tournament</classname>
    <activator>com.cubeia.test.systest.tournament.Activator</activator>
</tournament-definition>
```

The above example is the system integration test tournament included in Firebase. The following elements are used in the descriptor:

- “tournament-definition” - Mandatory root element. The “id” attribute specifiec a unique ID for the tournament *type* within Firebase and must be unique across all deployed tournament archives.
    - “name” - Mandatory. Simple tournament name, most used for logging.
    - “version” - Mandatory. Simple tournament version, most used for logging.
    - “classname” - Mandatory. Fully qualified class name of the MTT logic class for this tournament.
    - “activator” - Mandatory. Fully qualified class name of the tournament activator.

Firebase will read the tournament descriptor when it starts, and create the tournament logic and activator accordingly.

Deployment

When a tournament is packaged, it can be *deployed* in a Firebase server. This is simply the process of copying the TAR file to the `game/deploy` folder of a Firebase installation. Please see the section on game deployment regarding hot-, and redeploy.

MTT Support

Firebase includes an `MTTSupport` class developers can extend instead of implementing `MTTLogic` directly. This is an abstract base class that contains support methods for seating and moving players and creating and closing tables.

Service Development

What is a Service?

A service is a software module, deployed and instantiated as a singleton on a Firebase server. All services together form the service stack on single servers. Services are local to a server, as opposed to games, or their container nodes. There is no built in support for distribution, clustering or remote service control. The services are treated as first class members in a server, and are prerequisites without which the server will refuse to start. If any service reports an error at startup the server will halt immediately.

> **Note:** A deployed service is guaranteed to be started *before* any game and to be available during the lifetime of the server. Furthermore, Firebase will refuse to start if *any* service reports error on startup.

A service in Firebase consists of at least two classes, one interface which extends the `Contract` interface, which exposes the methods a service provides, and is usually called "the service contract", and one class which implements the the service contract and also implements the `Service` interface.

The Firebase server comes with a number of built in services which are not accessible from the games, but also some public services for accessing data sources, the transaction manager, etc. Developers can implement their own services to provide common functionality between the games and these will always to "public" within a server.

A service is accessed via a service registry. The registry is a simple map of services where each service is associated with a globally unique string acting as a public ID, as well as their contract.

Services can depend on each other, and will be initiated in dependency order such that dependent services will be started after the services they depend on. The server will detect circular dependencies and refuse to start if one is found.

A deployed service is "isolated". This means it will be loaded by its own class loader and manage its own resources. It may, however, export classes which consequently can be used by the entire server.

Proxy Services

From Firebase version 1.9 and onwards it is possible to implement a service contract as a Java Reflection dynamic proxy. This allows you to wrap any method call with common functionality, load the service via IOC frameworks etc.

Contract and Implementation

A service is logically described by two classes, its contract interface and its concrete implementation. The contract interface must extend the marker interface `Contract` and the implementation of the service must not only implement the contract but also implement the interface `Service` which adds lifetime and control methods for the service.

As a result, writing a service starts with defining two classes, the public interface (which extends `Contract)` and a concrete class with implements the public interface and the `Service` interface. For example:

```
public interface HelloWorldService extends Contract {

  /**
   * Say 'hello world' on standard out.
   */
  public void sayHello();

}
```

The above *contract* specifies the service methods that will be exposed to the rest of the server. This contract should be accompanied by an implementation of both `HelloWorldService` and `Service`, for example:

```
public class HelloWorldServiceImpl implements HelloWorldService, Service {

  @Override
  public void sayHello() {
    System.out.println("Hello World!");
  }

  [...]
}
```

In the above code, the lifetime methods of the service implementation have been removed.

A service is only accessed by games or other services by its contract or public ID. The service lifetime methods are used only by Firebase for controlling the service. In fact, client using a service should not assume that the implementation of the service contract is at all the same instance as their implementing class, the platform may insert dynamic proxies or similar measures to provide additional functionality.

Service Lifecycle

A service implementation must implement the interface `Service.` This interface contains the service lifecycle methods that will be used by Firebase to initiate, start, stop and finally destroy the service. This is reflected in the interface which slightly abbreviated looks like this:

```
public interface Service extends Initializable<ServiceContext> {

  public void init(ServiceContext con) throws SystemException;

  public void start();

  public void stop();

  public void destroy();

}
```

The services are handled by the service registry. If the `init` method fails the entire server will stop. However, there is no such provision for the start method. In other words, a service must make sure its prerequisites are met during the `init` method because it will not get another chance to halt the execution of the server.

The service lifetime methods are not reentrant. Each method is guaranteed to be called once only. The service will be initiated together with all other service in the stack, in dependency order, before the rest of the server, including the games, are even loaded. However, services are lazily started only when accessed the first time.

Configuration

A service has three main methods of configuration:

1. Through the Firebase cluster configuration.
2. Through the files in the server configuration directory. The services can access this directory via the `ServiceContext` and are free to read files from it at their discretion.
3. Through files packaged in the services archive. These files can be accessed via a helper interface called `ResourceLocator` which is available through the ServiceContext.

For example, a service may ship default configuration within the service archive and then use property files in the server configuration directory to override the default values.

The Service Registry

Services are accessed via the *service registry*. The registry handles the service deployment, their context and their lifetime. Services gets a reference to their containing registry via the `ServiceContext` and the games via their corresponding `GameContext`.

The set of all services on a server is called the *service stack*. Due to the dynamic nature of a Firebase cluster, it is assumed by the platform that the service stack is uniform across a cluster of several servers. In other words: all services should be deployed on all servers.

Services are accessed from the registry via their contract interface or via their public ID. In a majority of cases a lookup on the service contract is the normal case. Should the contract not be found, null is returned. For example:

```
ServiceRegistry registry = // get reference here...
Class<?> key = MyServiceContract.class;
MyServiceContract service = registry.getServiceInstance(key);
```

If more than one service implements the same interface, which will be unusual, the above method returns the first instance it finds. In such cases the public ID of a service must be used if it is significant which implementation is used.

Receiving and Sending Events

Although services are not normally a part of the event flow within a Firebase cluster, they can be receivers of events. The events can be sent from clients to a particular service, identified either by the contract class or by the public id. The service marks itself eligible for receiving events by implementing the `RoutableService` interface:

```
public interface RoutableService {

  public void setRouter(ServiceRouter router);

  public void onAction(ServiceAction e);

}
```

The 'onAction' method works much as the 'onMessage' method in message driven enterprise Java beans. Implementations must be able to handle messages asynchronously. The RoutableService interface can be implemented on the service class directly, the service contract does not need to extend it.

Via the routable service interface the service is given a reference to a ServiceRouter, which is the service's access point to the underlying Firebase message bus. As opposed to ordinary dependencies, the outgoing router is set via a usual 'setter' method and not via the service context. This is because a routable service implicitly depends on the Firebase message bus, which in itself is a service. The setRouter(ServiceRouter) method will be invoked after initialization but before starting.

The service router looks like this:

```
public interface ServiceRouter {

  public void dispatchToService(
    ServiceDiscriminator disc,
    ServiceAction action) [...];

  public void dispatchToPlayer(int playerId, ServiceAction action);

  [...]

}
```

It is worth noticing that both dispatch methods act asynchronously. In other words, even if the action to dispatch is for a service on the local stack, it will first be handed off to a separate thread before delivery.

Packaging

Services are deployed on a server in units called SAR (service archive) files. These files are compressed using a standard ZIP compression and contain a mandatory common structure.

Each service is packaged in their own unit, in other words there can be only one service per SAR file.

Service Descriptor

Within a SAR file a service must be declared using a service descriptor. This is an XML file which describes the service the archive contains. Below is an example service descriptor, containing all legal elements:

```
<service auto-start="false">
  <!-- Mandatory name of service -->
  <name>A Simple Example Service</name>
  <!-- Mandatory UID/namespace of service -->
  <public-id>ns://www.cubeia.com/examples/service</public-id>
  <!-- Mandatory FQN of contract class, 1 or more allowed -->
  <contract>com.cubeia.example.service.SimpleService</contract>
  <!-- Mandatory FQN och implementation class -->
  <service>com.cubeia.example.service.impl.ServiceImpl</service>
  <description>This is a simple example service.</description>
  <dependencies />
  <exported>
    <!-- Export all classes in a package -->
    <package>com.cubeia.example.service.*</package>
    <!-- Export single class -->
    <class>com.cubeia.example.service.error.MyException</class>
  </exported>
</service>
```

The service descriptor must be named 'service.xml' and be placed in a 'META-INF' folder in the root of the service archive. In other words:

```
/META-INF/service.xml
```

Of the elements in the example above, 'name', 'public-id', 'contract' and 'service' are mandatory. The name should a readable title for the service. The public ID is a globally unique string identifier, normally modelled on a namespace, and is used to identify the service. The contract should detail the service contract interface class (fully qualified class name). And finally the service should detail the fully qualified class name of the implementing service class.

Structure

The SAR file follows a simple structure. The service and its associated classes should be packaged in a JAR file and placed in the SAR file root. The name of the JAR file is inconsequential. The service descriptor should be placed in a "META-INF" directory and external libraries in the "META-INF/lib" directory. The structure should match:

```
/*.jar
/META-INF/service.xml
/META-INF/lib/*.jar
```

Dependencies

Services may depend on each other. In practical terms this only means that depending services will be initiated after their dependents. As a result it is safe for a depending service to reference its dependent during its own initiation.

Dependencies are declared using the service descriptor 'dependencies' element. This element accepts child elements 'contract' or 'public-id'. For example:

```
[...]
  <dependencies>
    <contract>com.cubeia.example.SimpleService</contract>
    <public-id>my-second-service</public-id>
  </dependencies>
[...]
```

The above declaration explains that the declared service depends on any service implementing the SimpleService contract and the service declared with the public id 'my-second-service'. Circular dependencies are not allowed.

Exporting Classes

Services are *isolated*. This means that services will be loaded by their own class loader and manage their own resources. However, if this would only be the case services would not be usable as they could not be referenced from classes loaded by other class loaders. To be usable by games or other services, a service *exports* classes or entire packages. An exported class is usable by the entire server.

The service contract interface is always exported.

Exported classes or packages are declared using the 'exported' element and its child elements 'class' and 'package'. The 'class' element should details a fully qualified class name of a class to export. The 'package' element is used to export entire packages and optionally their sub-packages as well. It must end with either a hyphen or a star, where a package name ending with '*' represents the current package but no sub-packages and a '-' means the current package and all sub-packages. For example:

```
[...]
  <exported>
    <class>com.cubeia.example.SimpleClass</class>
    <package>com.cubeia.example.common.*</package>
    <package>com.cubeia.util.-</package>
  </exported>
[...]
```

The above example exports the SimpleClass. It also exports the 'com.cubeia.example.common' package but none of its sub-packages. And finally it exports the 'com.cubeia.util' package, including all its sub-packages.

Please refer to the class loading section in this manual for more information on exported classes.

External Libraries

Services may use external libraries. These can either be placed globally in the server, or be packaged in the service SAR file.

The entire server loads JAR files from the 'lib/commons' folder in the server installation directory. Consequently, services may place required libraries there. However, this is strongly discouraged as this (1) increases the maintenance burden on the server; (2) decreases service isolation; (3) invalidates future hot re-deployment of services; and (4) forces the given library on the entire server with possible class loading clashes as a result.

Dependencies can also be shared if they are placed in 'game/lib' which is common for all deployed artifacts.

Finally JAR files can also be embedded in the SAR, under '/META-INF/lib'.

Archive Access

Service may read resources from their own SAR file. This is done via the `ResourceLocator` module which is made available to them in their `ServiceContext`.

Deployment

When a service is packaged, it can be *deployed* in a Firebase server. This is simply the process of copying the SAR file to the `game/deploy` folder of a Firebase installation. Note that services will be loaded before any game or tournament archives.

Proxy Services

Instead of the concrete service implementation described above it is possible to implement the service contract as a dynamic proxy. For example, if a service contract looks like this...

```
public interface EchoService extends Contract {

    public String echo(String msg);

}
```

… you can actually implement it as a dynamic proxy invocation handler:

```
public class ServiceHandler implements InvocationHandler {

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
        if(method.getName().equals(“echo”)) {
            return args[0];
        } else {
            throw new IllegalStateException(“Unknown method: “ + method.getName());
        }
    }
}
```

The immediate benefit of this is that it opens for IOC creation of a concrete implementation as well as commons cross cutting concerns to be applied.

Packaging and deployment

A proxy service is packaged and deployed as a normal service with one exception: Instead of having a “service” element in the descriptor you supply a “invocation-handler” instead. For example:

```
<service auto-start="false">
```

```
  <!-- Mandatory name of service -->
  <name>A Simple Proxy Service</name>
  <!-- Mandatory UID/namespace of service -->
  <public-id>ns://www.cubeia.com/examples/service</public-id>
  <!-- Mandatory FQN of contract class, 1 or more allowed -->
  <contract>com.cubeia.example.service.SimpleService</contract>
  <!-- Mandatory FQN och implementation invocation handler -->
  <invocation-handler>com.cubeia.example.service.impl.ServiceHandler</invocation-
handler>
  <description>This is a simple example service.</description>
  <dependencies />
  <exported />
</service>
```

Firebase Services

The majority of the Firebase services are internal and will never be accessed by games and game related services. A subset of them however, are public and intended to be used and accessed externally. Services are accessed via the service registry, which is available in most component contexts. Games will access the service registry through the `GameContext` class, services through the `ServiceContext`, etc.

Services are normally referenced by their contract class. For example, given the service "my service" which has the contract interface `MyServiceContract` it will be accessed via the service regisry like so:

```
ServiceRegistry reg = // get registry here...
Class<MyServiceContract> key = MyServiceContract.class;
MyServiceContract service = reg.getServiceInstance(); // get instance
```

Public Services

Cluster Config Provider

The cluster configuration provider is an accessor service for the cluster wide configuration. Firebase keeps two sets of configuration properties, one which is local to each server ('server properties', see below) and one which is configured at the master node and propagated across the cluster ('cluster properties').

The provider can be used to access the raw properties, but also to fulfil so called configuration interfaces. See Using the Cluster Configuration for more details.

- Contract: ClusterConfigProviderContract

Server Config Provider

This service contains the configuration properties for a single server ('server properties'). These properties are local to each server and may be different across the cluster.

The provider can be used to access the raw properties, but also to fulfil so called configuration interfaces. See Using the Cluster Configuration for more details.

- Contract: ServerConfigProviderContract

Datasource Service
The data source service provides access methods for all deployed data sources. Deployment of data sources is described in the data source deployment section.

- Contract: DatasourceServiceContract

System State
A given cluster shares a tree-like data structure called the system state. This state contains the lobby, mapping between players and the tables they sit at, and much more. Advanced services may need to interact with the system state through this service.

- Contract: PublicSystemStateService

System Transaction Provider
Firebase uses JTA to manage user transactions per event and game. This service gives access to the current user transaction object but also to the underlying transaction manager for system level transaction integration.

- Contract: SystemTransactionProvider

Client Registry Service
This is the publicly available client registry which manages all connected clients and sessions. The registry gives access to session details for logged in clients.

- Contract: PublicClientRegistryService

Service Router
Normally services adopts a request response pattern by implementing `RoutableService`, however, this service exposes a service router externally which can be used to send event to clients etc.

- Contract: RouterService

Denial of Service Protection
The denial of service (DOS) protector service is a simple service which contains server-wide frequency access rules. It can be used for rudimentary DOS control.

- Contract: DosProtector

Simple Usage

1. For a given access which is to be defended, decide on a string identifier, for example 'chatSpamRule'. This identifier will be used in configuration and in access control. All identifiers starting with an "_" are reserved for internal Firebase purposes.

2. On startup, configure the DOS protector for the access. This is done via the service interface, using so called 'rules'. There is a common frequency access rule which can be used for most purposes. For example, in semi-code:

```
DosProtector dos = // lookup service
// At most 2 accesses per 1000 millis
FrequencyRule rule = new FrequencyRule(2, 1000);
dos.config("chatSpamRule", rule);
```

3. At each access point, use the DOS protector to check if a given access should be granted. For example:

```
Integer playerId = // player to check spam for
DosProtector dos = // lookup service
if(!dos.allow("chatSpamRule", playerId)) {
  // Continue as normal
} else {
  // Drop message
}
```

Plugin Services

Firebase has a number of services which are defined as 'plugin' services which have their contract already defined in the Firebase API. These are typically used to extend core server behaviour; in particular: a component or service may have a default behaviour which can be overridden by a plugin service.

In order to implement a plugin service you build a service and have it implement the service *contract* of the plugin service. Drop in your service archive in the regular deploy folder in Firebase and it will be used, no special configuration is needed.

If a specific plugin service is not found, Firebase reverts to its default behaviour. The behaviour of a component that uses a plugin service usually looks like this:

1. Initialize
2. On first call check if a plugin service is deployed:
   a. if it is: use it for the desired behaviour
   b. if it is not: use default behaviour
3. On subsequent calls, repeat #2.
4. Destroy

For example, the Firbease login components uses the login locator to implement authentication. If a login locator is implemented and deployed all authentication requests will be firwarded to it, if not Firebase will asume that the login name is also the screen name and that the password is an integer which is also the player ID.

Login Locator

The login locator extends Firebase to provide authentication for an installation. Out of the box Firebase assumes that the player password is an integer and will use it as the player ID, and also allow all login requests. If a login locator is deployed it will handle the authentication instead.

- Plugin Contract: LoginLocator

Writing a LoginHandler

Firebase supports a plugin model of login handling. Since a network may have many operators that have different ways of validating a user, Firebase also supports multiple login handlers. These are managed by the login locator plugin service. For example:

```
public class TestLoginLocator implements Service, LoginLocator {

  private TestLoginHandler handler = new TestLoginHandler();

  [...]

  @Override
  public LoginHandler locateLoginHandler(LoginRequestAction request) {
    // Here we could use different login handlers for different
    // operators, but for simplicity we're restricting ourselves
    // to one unified login handler
    return handler;
  }
}
```

The `TestLoginLocator` returns a `LoginHandler`, so we need to create on of those as well. The responsibility of the `LoginHandler` is to authenticate the user, then verify or deny the login request.

```
public class TestLoginHandler implements LoginHandler {

  private AtomicInteger pid = new AtomicInteger(0);

  @Override
  public LoginResponseAction handle(LoginRequestAction req) {
    // At this point, we should get the user name and password
    // from the request and verify them, but for this example
    // we'll just assign a dynamic player ID and grant the login
    return new LoginResponseAction(true,
        req.getUser(),
        pid.incrementAndGet());
  }
}
```

In order to use this new service we need to package and deploy it so that Firebase can use it. We don't need to configure Firebase further as this is a so called 'plugin service', meaning that Firebase will use it automatically if it is deployed.

Post Login Processor

The post login processor lets you listen for client login events.

- Plugin Contract: PostLoginProcessor

The events that are reported are specified in detail in the Javadocs but are typically login, logout and disconnect. Listening for logout and disconnects allows you to clean up any data associated with that user that might be stored in memory.

Player Lookup

The player lookup service enables connection clients to query Firebase about other players. A query request is created by a client and sent to the server, and if a player lookup service is deployed it will be used to create a response.

- Plugin Contract: PlayerLookupService

This is a plugin service, meaning the contract is included in the Firebase API, and that it only need to be deployed in order to be used by the system.

Local Handler

The local handler service is a service that can receive data from a client that is *not logged in*. Routable services need a logged in client since the player/client id is the unique identifier used for routing events throughout the system. There can only be only local handler service deployed, so if you need to send different events you will need to implement a dispatcher.

- Plugin Contract: LocalHandlerService

A local handler can only be called from connections that is local to the server that it is deployed in.

Below is an example of the java implementation of a simple local handler implementation. Code that is not directly related to the implementation specifics of the service has been omitted:

```
public class SimpleLocalService implements
  LocalHandlerService, Service {

  [...]

  public void handleAction(
    LocalServiceAction action,
    LocalActionHandler loopback) {

    String text = new String(action.getData());
    String resp = text.toUpperCase();
    int seq = action.getSequence();
    LocalServiceAction response = new LocalServiceAction(seq);
    response.setData(resp.getBytes());
    loopback.handleAction(response);
  }
}
```

Chat Filter

The chat filter plugin service can be used to filter or modify chat messages server-side. The filter service will be instantiated on each server node in a cluster, this gives rise to a possibly unexpected behaviour (see below). The main use for this service should be to filter messages.

- Plugin Contract: ChatFilter

The notification of creation and destruction of a chat channel is not "atomic across multiple virtual machines". In effect, this means a multi-server cluster may get notifications of creation or destruction simultaneously on two separate server. No attempt to synchronise this is made by Firebase itself, and as such implementers must be aware that this may happen in some circumstances.

The message flow for an incoming chat message in Firebase looks like this:

1. Chat message arrives from client at server.
2. The local chat component looks for a `ChatFilter` implementation among the services.
3. If a service is found in #2, it is invoked before any relevant action is taken.
4. If a message proceeds from the filter, it will distributed to chat channel listeners.

System Info Query

A client can query Firebase for system information by sending a `SystemInfoRequestPacket`. The response will be populated with some predefined data (e.g. player count), but it is also possible for a developer to append extra information to the response.

- Plugin Contract: SystemInfoQueryService

A system info request is generated by the client, the request is received and dispatched in the client node locally, i.e. the request will only be handled locally and will not be distributed in the system. The call to the system info service will be asynchronous by a different thread to make sure that any custom implementation is not blocking game packets from execution if the logic is stalling (e.g. database calls with high latency).

Below is an example of the java implementation of a simple lookup implementation. Code that is not directly related to the implementation specifics of the system info service has been omitted:

```
public class SystemInfoMutator implements
  Service, SystemInfoQueryService {

  private Random rng = new Random();

  /** Adding a randomly generated number as 'Jackpot' */
  public SystemInfoResponseAction appendResponseData(
    SystemInfoResponseAction action) {

    int next = rng.nextInt(100000);
    Parameter<Integer> p = new Parameter<Integer>("Jackpot", next, Type.INT);
    action.getParameters().add(p);
    return action;
  }

  [...]

}
```

Persistence

Firebase supports the deployment of JDBC datasource which can be configured to be used in a JTA context or using local transactions. From version 1.9 JPA archives are no longer supported, however JPA is perfectly possible to use within game, tournaments or services, it is just not supported as a separate archive.

JDBC

Firebase support databases via deployment of data sources as XML files in the server deployment folder. A data source is a basic JDBC connection to a database. All deployed datasources setup by Firebase will be pooled for performance.

Deployment and Configuration

To deploy a data source, specify the connection properties in an XML file and place it in the deployment folder of the server. The name of the file should be of the following pattern: ‘<name>-ds.xml’ where ‘<name>’ will be the name the data source will be registered as.

A simple data source config file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <!-- Mandatory JDBC URL -->
  <entry key="url">jdbc:mysql://localhost:3306/test</entry>
  <!-- Mandatory JDBC driver class -->
  <entry key="driver">com.mysql.jdbc.Driver</entry>
  <!-- Mandatory database user name -->
  <entry key="user">root</entry>
  <!-- Mandatory database user password -->
  <entry key="password"></entry>
</properties>
```

If the data source file above is contained in a file with the name ‘foo-ds.xml’, then you can reference the data source in Firebase by the name ‘foo’. For example, via the data source service:

```
ServiceRegistry reg = // get registry via context
Class<DatasourceServiceContract> key = DatasourceServiceContract.class
DatasourceServiceContract service = reg.getServiceInstance(key);
DataSource src = service.getDatasource(“foo”);
```

The ‘tx-type’ property in the data source file specifies the data source transaction type. This can be used to setup XA data sources.

- “tx-type” - Data source transaction type - [NON-TX | LOCAL-TX] - default: NON-TX - Optional.

Available values are:

- “NON-TX” - Non-XA data source, vanilla JDBC connection pooling. This is the default.
- “LOCAL-TX” - XA data source, the connection will participate in the event user transaction.

For XA data sources, the JDBC vendor must provide an XA compatible data source interface. However, it is possible to emulate XA compatibility over standard JDBC drivers, but this is sub-optimal and should be avoided if possible.

Default Properties (NON-TX)

These properties are for data sources with no ‘tx-type’ set, or ‘tx-type’ set to ‘NON-TX’.

- “driver” - Database driver class name - Mandatory.

- “url” - Database connection URL - Mandatory.
- “user” - Database user name - Mandatory.
- “password” - Database user password - Mandatory.
- “min-pool-size” - Minimum pool size - default: 2 - Optional.
- “max-pool-size” - Maximum pool size - default: 10 - Optional.
- “validation-statement” - Statement used to check connection validity before use - Optional.
- “ttl” - Time to live for idle connections in seconds - default: 240 - Optional.

Using the above properties, Firebase can create a connection pool data source for the given driver. If "validation-statement" is set, the pool will check all connections prior to use. Below is an example configuration for a MySQL data source:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="url">jdbc:mysql://localhost:3306/test</entry>
  <entry key="driver">com.mysql.jdbc.Driver</entry>
  <entry key="user">user1</entry>
  <entry key="password">password1</entry>
  <entry key="validation-statement">SELECT 1</entry>
  <entry key="min-pool-size">2</entry>
  <entry key="max-pool-size">20</entry>
</properties>
```

XA Properties (LOCAL-TX)

These properties are used to configure an XA data source.

- “xa-data-source” - JDBC vendor XA data source class name - Mandatory.
- “pool-size” - Fixed pool size - default: 5 - Optional.
- “ttl” - Time to live for idle connections in seconds - default: 240 - Optional.
- “ds.*” - Data source specific properties.

In order to initiate the vendor XA data source a set of properties is matched against the data source. This matching is done using standard bean introspection, for example property 'url' will be matched to method 'setUrl' etc. These properties are specified in the configuration using the prefix 'ds.', for example 'ds.password', 'ds.user' etc. The connections will be checked for validity every 'ttl' second.

Currently the local TX pooling uses Connection.getMetaData() to validate the connections. This slightly naive implementation can be slow on certain databases.

Below follows an example using Oracle 1.2.0.10 XA data source on a fictional database:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="tx-type">LOCAL-TX</entry>
  <entry key="xa-data-source">
   oracle.jdbc.xa.client.OracleXADataSource
  </entry>
  <entry key="pool-size">50</entry>
  <!-- Data Source Properties -->
  <entry key="ds.url">jdbc:oracle:thin:@localhost:1521:XE</entry>
  <entry key="ds.password">password1</entry>
  <entry key="ds.user">user1</entry>
</properties>
```

Emulated XA Transaction

If the JDBC vendor does not provide a usable XA data source Firebase can emulate XA functionality with some restrictions. This is indicated in the config by setting ‘tx-type’ to ‘LOCAL-TX’ but replacing 'xa-data-source' property with a 'driver' property. Firebase will then proceed to wrap the JDBC driver and emulate the functionality. The connections will be checked for validity every 'ttl' seconds.

Currently the local TX pooling uses Connection.getMetaData() to validate the connections. This slightly naive implementation can be slow on certain databases. Please contact Cubeia Ltd if this is a problem.

- “driver” - Database driver class name - Mandatory.
- “url” - Database connection URL - Mandatory.
- “user” - Database user name - Mandatory.
- “password” - Database user password - Mandatory.
- “pool-size” - Fixed pool size - default: 5 - Optional.
- “ttl” - Time to live for idle connections in seconds - default: 240 - Optional.

Below is an example of an emulated MySQL data source:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="tx-type">LOCAL-TX</entry>
  <entry key="driver">com.mysql.jdbc.Driver</entry>
  <entry key="pool-size">10</entry>
  <entry key="url">jdbc:mysql://localhost/test1</entry>
  <entry key="password">password1</entry>
  <entry key="user">user1</entry>
</properties>
```

Access via JNDI

Data sources that have been deployed can be accessed via JNDI. For example, if the data source is named “my-test”, it can be accessed like so:

```
InitialContext con = new InitialContext();
DataSource ds = (DataSource) con.lookup(“java:comp/env/jdbc/my-test”);
```

This makes it possible to configure JPA persistence archives with deployed data sources.

JPA

From Firebase version 1.9 JPA is no longer supported as separate deployable units. However, it is perfectly possible to use JPA with any game, service or tournament.

Transactions

Firebase can be configured to add a JTA transaction to each processed event. The transaction is used to optimise network performance for state transfers and and ensure atomic event processing. Developers can integrate their code with the current user transaction on several levels, by using XA data sources, by using JTA persistence configurations or by manually enlisting XAResources with the system transaction manager.

The current transaction behaves much like CMT (container managed transactions) in J2EE. A UserTransaction will begin when an event arrives at a game node, and will be committed or rolled back when the event processing is finished. Currently there is no support for opting out of the transaction.

Firebase utilises a JTA implementation from Bitronix

Known Limitations

Database and JDBC driver support for XA is somewhat lacking. Please make sure your choice of database supports XA properly before integrating. Firebase can emulate XA for JDBC drivers, but for complex scenarios this may not be enough. A partial investigation of databases and XA support have been made by the Bitronix team here.

Firebase does not support XA recovery.

Firebase does not support distributed XA transactions.

JTA

XA Data Sources

Data sources can be configured as XA data sources in which case they automatically will be part of the event processing user transaction. Below is an example data source configuration for XA integration (somewhat strangely formatted to fit page):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="tx-type">LOCAL-TX</entry>
  <entry key="xa-data-source">
    oracle.jdbc.xa.client.OracleXADataSource
  </entry>
  <entry key="pool-size">50</entry>
  <!-- Data Source Properties -->
  <entry key="ds.url">
    jdbc:oracle:thin:@localhost:1521:XE
  </entry>
  <entry key="ds.password">password1</entry>
  <entry key="ds.user">user1</entry>
</properties>
```

JPA Persistence

Configuring the persistence manager to use the JTA user transaction during event execution is done as per the JPA specifications. By utilising a JTA (XA) data source, and setting the transaction type to "JTA" the entity manager will automatically participate in the event transaction. Below is an example configuration (somewhat strangely formatted to fit page):

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
      http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
      "
  version="1.0">
  <persistence-unit name="test" transaction-type="JTA">
    <jta-data-source>java:comp/env/jdbc/my-xa-data-source</jta-data-source>
    <class>com.cubeia.firebase.api.entity.User</class>
    <properties>
      <property
       name="hibernate.dialect"
       value="org.hibernate.dialect.OracleDialect"/>
      <property
       name="hibernate.hbm2ddl.auto"
       value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```

Hibernate and JTA

In order to use Hibernate with Firebase and JTA you will need to supply Hibernate with a JTA lookup manager. This is easily done by extending their AbstractJtaPLatform class:

```
public class FirebaseJtaPlatform extends AbstractJtaPlatform {

	public static final String TM_NAME = "java:comp/env/TransactionManager";
	public static final String UT_NAME = "java:comp/env/UserTransaction";

	@Override
	protected TransactionManager locateTransactionManager() {
		return (TransactionManager) jndiService().locate( TM_NAME );
	}

	@Override
	protected UserTransaction locateUserTransaction() {
		return (UserTransaction) jndiService().locate( UT_NAME );
	}
}
```

This will use JNDI to locate the transaction manager. You can then configure Hibernate to use the above JTA platform:

```
<properties>
    <property
        name="hibernate.transaction.jta.platform"
        value="mypackage.FirebaseJtaPlatform"/>
       ...
```

XA Resource

Advanced usage may include manual integration with the controlling `TransactionManager`. The transaction manager is available through the public system service `SystemTransactionProvider`, which also gives access to the `UserTransaction` object. An example of this kind of integration could be a accounting service utilised by games to manage player accounts, by enlisting with the system transaction manager the service can be reasonably sure of only committing if the entire event execution succeeded.

Further Reading

If XA integration is required developers are encouraged to investigate possible repercussions before system design. This is particularly true if `XAResource` integration is needed.

Some recommended reading:

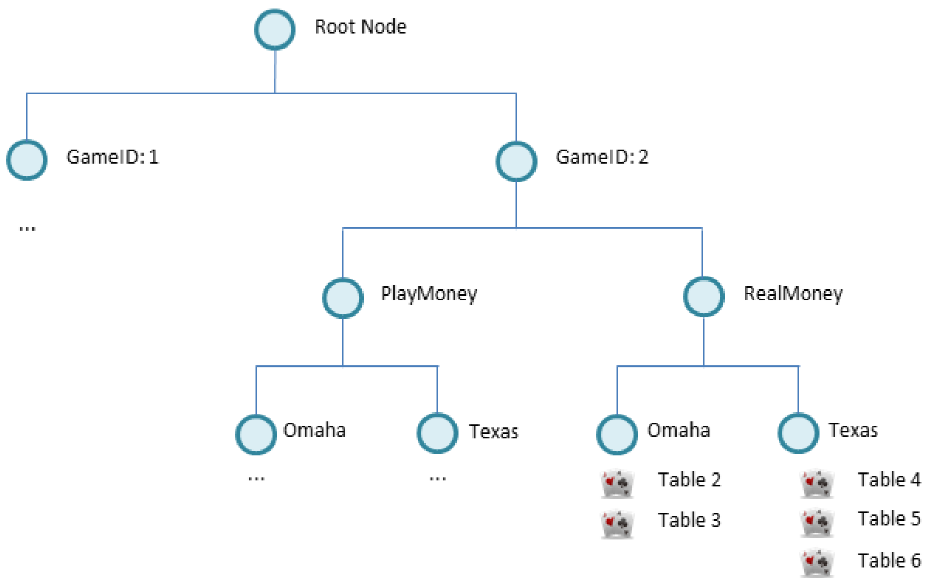
- The JTA Specification
- The XA Specification
- XA Exposed, pt. I, II and III

Lobby

Structure

The lobby data is modelled in the system as a tree structure. The top of the tree origins with the game id, is expanded down along user defined paths and has individual tables as leaf-nodes.

Below is an example of a lobby tree with two games deployed with id 1 and 2 respectively. Game with id 2 has an example tree structure for Texas Holdem and Omaha tables running play money and real money. The individual Tables are stored under each end node as leaves.



Lobby Path

The lobby path is the path in the lobby tree originating *from the game ID node*. For example:

- The root of game ID 2: “/”

- The root of the real money tables for game ID 2: “/RealMoney”
- The root of the “texas” tables for game ID 2: “/RealMoney/Texas“
- Table 5 for game ID 2: “/RealMoney/Texas/5“

The lobby path is always rooted at the game ID and will always end with a table or tournament ID. The following vocabulary applies:

- “type” - A lobby path is either a ‘table’ or ‘tournament’ type. This is usually transparent to the developer. Future types may include user space types.
- “area” - This is the *game ID* or a *tournament ID* and is always the root of the path.
- “domain” - The developer controlled path of the object.
- “object” - The *table ID* or *tournament instance ID.* This is an integer ID and is mandatory.

From the above follows that the developer really only controls the the ‘domain’ of of the lobby path. This is done when a table or a tournament instance are created. For example, in order to create a path for table 5 from the example above:

```
LobbyPath tablePath = new LobbyPath(2, “RealMoney/Texas”, 5);
```

The above creates a path in for game ID 2, with the domain ‘RealMoney/Texas’ and table ID 5. Note that the type of the path, ie ‘table’ or ‘tournament’, will be enforced by FIrebase automatically.

Lobby Attributes

Each leaf in the lobby structure, ie tables or tournaments, has a set of associated properties or attributes. These will be propagated to subscribing clients.

Every lobby attribute starting with ‘_’ are reserved for Firebase. For tables these are:

- “_CAPACITY” - Table size. Integer.
- “_ID” - Table ID. Integer.
- “_GAMEID” - Game ID. Integer.
- “_LAST_MODIFIED” - Timestamp in milliseconds for last modification. Date.
- “_MTT_ID” - If this is a tournament table this is the tournament ID, otherwise empty. Integer.
- “_NAME” - Table name. String.
- “_SEATED” - Number of seated players. Integer.
- “_WATCHERS” - Number of watching players. Integer.

These attributes will be enforced by Firebase. The lobby attributes can be of the following data types:

- String - UTF8 string value. This is the default for all attributes. All attributes will be serialized as strings in delta changes and lobby snapshots.
- Integer - 32 bit signed integer. Serialized as a string value.
- Date - Date as milliseconds since the epoch. Serialized as a string value.

Handling Lobby in the Server

The lobby is manipulated server side in two places:

1. When a table or tournament is created, its position in the lobby is determined by the ‘domain’ set by the developer.
2. When an action is executed at a table or a tournament the context has a ‘attribute accessor’ that can be used to set, or change attributes.

The domain is fixed when the object is created and can never be changed. It is created on table or tournament creation, for example, when a table is created a `TableCreationParticipant` is used, and contains a method for creating the initial path:

```
public LobbyPath getLobbyPathForTable(Table table) {
  return new LobbyPath(
    table.getMetaData().getGameId(),
    “RealMoney/Texas”,
    table.getId());
}
```

The above sets the domain of the table to "RealMoney/Texas”. A similar method exists for creating the initial path for tournaments.

When actions are executed the table or tournament, lobby attributes can be set and updated. For example, when executing an action on a table:

```
public void handle(GameDataAction action, Table table) {
  LobbyAttributeAccessor acc = table.getAttributeAccessor();
  acc.setIntAttribute(“my-counter”, counter++);
}
```

Note that all attributes can be read as strings, and will be serialized as such.

Handling Lobby in the Client

Subscribing

Delta Changes

The lobby supports delta changes, only changes to a table is sent out to the clients. The first data sent out to a client will be a complete snapshot of all tables that are in interest of the client. All successive updates to the lobby data will then be sent as delta updates.

Subscription Policy

Clients registers for subscribe/unsubscribe for a given lobby path. The client will subscribe to all tables contained under all sub-paths contained under that path.

A client that subscribes for a specific path will first receive a complete set of data for the patyh. This includes, but is not limited to, table name, table capacity etc. This is considered static data. Further updates will only consist of delta changes to the table data, e.g. how many are currently seated.

A client can unsubscribe at any point. This will cause the server to stop sending lobby data to the client. If the client re-subscribes to the same path, a full update will be sent to the client. The server will not keep track of the latest state sent for clients that has unsubscribed to a certain lobby key.

Miscellaneous

Using the Cluster Configuration

The Firebase cluster configuration is a properties file which is read by the master node and then propagated through the cluster. These properties are available through the cluster config provider service.

Firebase utilises a mapping mechanism between these properties and standard Java interfaces for ease of access and default values. This mechanism is documented in the Java API documentation for Configurable, but is also outlined below.

In order to utilise this configuration mechanism, this is what you'll do:

1. Create an interface extending `Configurable` with accessor methods for the configuration data. This interface will be proxied by Firebase to return values matched by the configuration properties. This interface must be global, ie. placed in 'lib/common' or exported by a service.
2. Decide on a 'namespace' for your configuration data. This namespace is used to keep track of different subsets of the configuration properties. For example, 'com.acme.wallet'.
3. Add properties to 'cluster.props' which matches you namespace and configuration interface. The method names are matched to properties much like an ordinary Java beans matching, but are also prefixed by the namespace. Some example might be:

```
com.acme.wallet.timeout -> MyConfigurable.getTimeout()
com.acme.wallet.remote-url -> MyConfigurable.getRemoteUrl()
```

4. You'll access the configuration via the cluster config provider service. The namespace for a configuration interface is normally set in the `Configurated` annotation on the interface but can be overridden in this method.

The return types of the configuration interface is limited to the following types:

- Any primitive.
- String.
- InetAddress.
- Any class with a constructor taking a single string argument.

- Enums (by string matching)
- StringList (separated by commas)

Below follows an examples for a fictional service which needs an integration URL and a connection timeout configured. First, create the configuration interface as per #1 above.

```
@Configurated(namespace=”service.integration”)
public interface ServiceConfig extends Configurable {

  @Property(defaultValue=”60000”)
  public long getTimeout();

  @Property(defaultValue=”http://localhost:8080/test”)
  publi URL getIntegrationUrl();

}
```

The “Property” annotations above specifies the default values, so if no properties are added in the cluster configuration, the above values will be returned. The “Configurated” annotation adds a default namespace.

Next add properties to ‘conf/cluster.conf’:

```
service.integration.timeout=80000
service.integration.integration-url=http://foo/operatorOne
```

Now compare the interface with the properties. Firebase will try to match the namespace and method name from the interface with a property.

When you need the configuration, use the config provider to create an instance of your configuration interface, like so:

```
ServiceRegistry reg = // get registry from context
Class<ClusterConfigProviderContract> key = ClusterConfigProviderContract.class;
ClusterConfigProviderContract provider = reg.getServiceInstance(key);
ServiceConfig config = provider.getConfiguration(ServiceConfig.class, null);
// do something with your configuration here...
System.out.println(config.getIntegrationUrl());
```

As you can see above, the cluster config provider returns an instance of the configuration interface. This instance is a proxy and will read its return values from the properties file deployed on the Firebase master server.

Unified Archives

For simplicity it is possible to package multiple deployment units together as a single archive. This is called a unified archive (UAR) and should have the file extension ".uar". This archive should be a ZIP compression file and may contain anything that normally goes into the deployment folder except other UAR files.

Exploded Deployment

The UAR does not have to be a ZIP archive. For rapid deployment it can be a folder instead, however, the folder name must still end with ".uar".

Shared Class Loader

Deployment units within a unified archive will use a shared class loader making it possible to share classes between for example tournaments and games. It is also possible to place utility libraries directly within the UAR file itself to have them shared between the deployment units, at the following locations:

```
/*.jar
/META-INF/lib/*.jar
```

Services

Services deployed as members of a unified archive will lose their internal isolation as a result of the shared class loader. In other words, even though the implementation classes of a service may not be exported they can still be seen and used by other members of the same UAR. This usage, however, is discouraged and the services will still be isolated in respect to the rest of the system outside the UAR.

Example UAR Content:

```
/fungame.gar
/fungame-ga.xml
/shootout-tournament.tar
/shootout-tournament-ta.xml
/maindb-ds.xml
/utils.jar
/wallet.sar
```

Failure Detection

In order to detect failed network connections Firebase can be configured to use a ping mechanism from the server to the clients. In short the server will determine if a client is deemed 'idle' and start pinging it, and one more more missed ping responses will disconnect the client.

If failure detection is enabled, given client session 'X':

- For each received packet belonging to X, update time stamp X.y
- If (X.y + (initial-ping-delay)) > (system-time), start pinging the client
- For each ping:
    - Wait <ping-timeout>
    - If ping timeout:
        - If (ping-no) >= (failure-threshold), disconnect
        - Else log warning and schedule new ping
    - Else if ping return, schedule new ping
- If X receives data (other than ping), cancel ping if appropriate

Configuration

The following properties can be set in the cluster properties (shown with default values):

```
service.ping.ping-enabled=false
service.ping.initial-ping-delay=20000
service.ping.ping-interval=5000
service.ping.ping-timeout=3000
service.ping.failure-threshold=1
service.ping.number-of-threads=1
```

Intervals are configured in milliseconds, however some of them may be truncated down to seconds depending on underlying transport mechanism. It is recommended to configure the service in even seconds only.

In order to enable ping failure detection, set 'ping-enabled' to 'true'.

A client will be deemed 'idle' if the server does not get any data from it within 'initial-ping-delay' milliseconds. By keeping this value a multiple of the ping interval an installation can dramatically decrease the actual number of pings sent.

The frequency of the pings is configured by 'ping-interval' in milliseconds.

A ping will be considered failed after 'ping-timeout' milliseconds, and when the number of failed pings reaches 'failure-threshold' the client will be disconnected.

If one server handles very many clients, you may want to increase the number of threads used for scheduling purposes by setting 'number-of-threads', although this should rarely be needed.

Handling Pings In the Client

The Firebase Client API automatically handles ping messages on the client side. Developers using their own connection code needs to handle the `PingPacket` by simply returning it as soon as it is received. The packet must not be modified on the client side as it contains a unique id which will need to be preserved for the entire ping round-trip.

Building With Maven

This section talks about the Maven supports shipped with Firebase by default. For more information please refer to our Wiki.

Repository

In order to utilise the Cubeia Firebase Maven tools, you need the following repository added to your build configuration.

```
<repository>
  <id>cubeia-nexus</id>
  <url>http://m2.cubeia.org/nexus/content/groups/public</url>
  <releases>
    <enabled>true</enabled>
  </releases>
  <snapshots>
    <enabled>true</enabled>
  </snapshots>
</repository>
```

Archetypes

There are three main archetypes for Maven and Firebase, for creating GAR, SAR, TAR and UAR projects. The UAR archetype will setup a collection of the other three.

Game Archetype

To create a new game (GAR) module, execute:

```
`mvn archetype:generate \
  -DarchetypeRepository=http://m2.cubeia.com/nexus/content/groups/public \
  -DarchetypeGroupId=com.cubeia.firebase.tools \
  -DarchetypeArtifactId=firebase-game-archetype \
  -DarchetypeVersion=1.8`
```

Service Archetype

To create a new service (SAR) module, execute:

```
`mvn archetype:generate \
  -DarchetypeRepository=http://m2.cubeia.com/nexus/content/groups/public \
  -DarchetypeGroupId=com.cubeia.firebase.tools \
  -DarchetypeArtifactId=firebase-service-archetype \
  -DarchetypeVersion=1.8`
```

Tournament Archetype

To create a new tournament (TAR) module, execute:

```
`mvn archetype:generate \
  -DarchetypeRepository=http://m2.cubeia.com/nexus/content/groups/public \
  -DarchetypeGroupId=com.cubeia.firebase.tools \
  -DarchetypeArtifactId=firebase-tournament-archetype \
  -DarchetypeVersion=1.8`
```

Project Archetype

To create a new multi-module (UAR) project, execute:

```
`mvn archetype:generate \
  -DarchetypeRepository=http://m2.cubeia.com/nexus/content/groups/public \
  -DarchetypeGroupId=com.cubeia.firebase.tools \
  -DarchetypeArtifactId=firebase-project-archetype \
  -DarchetypeVersion=1.8`
```

Packaging

The Firebase archive plugin handles packaging for Firebase games, services, tournaments and unified archives. The normal Maven packaging rules apply, with the exception of the archive deployment descriptors, which should be placed in a subfolder of the 'resources' folder, like so:

- Game archives (GAR): `src/main/resources/firebase/GAME-INF/game.xml`
- Service archives (SAR): `src/main/resources/firebase/META-INF/service.xml`

- Tournament archives (TAR): `src/main/resources/firebase/META-INF/tournament.xml`

The contents of the 'src/main/resources/firebase' folder will be included in the root of the created archive.

Build Plugin

The Firebase archive plugin should be included in the build section of the POM, like so:

```
[...]

<build>
  <plugins>
    <plugin>
      <groupId>com.cubeia.firebase.tools</groupId>
      <artifactId>archive-plugin</artifactId>
      <version>1.8.0</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>

[...]
```

Packaging Type

The Firebase archive plugin recognises the 'firebase-sar', 'firebase-gar', 'firebase-tar', and 'firebase-uar' as packaging in the POM. For example:

```
<packaging>firebase-gar</packaging>
```

Example Game POM

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.my-company.test</groupId>
  <artifactId>funkyGame</artifactId>
  <packaging>firebase-gar</packaging>
  <name>Funky Game</name>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>com.cubeia.firebase</groupId>
      <artifactId>firebase-api</artifactId>
      <version>1.8.0-CE</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>com.cubeia.firebase.tools</groupId>
        <artifactId>archive-plugin</artifactId>
        <version>1.8.0</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

Running Firebase

In order to run your artifact from the command line and within Maven you need to add the following build plugin to your POM:

```
[...]

<plugin>
  <groupId>com.cubeia.tools</groupId>
  <artifactId>firebase-maven-plugin</artifactId>
  <version>1.9.0-CE-RC.1</version>
  <configuration>
    <deleteOnExit>false</deleteOnExit>
  </configuration>
</plugin>

[...]
```

When the above plugin is added, you can run Firebase with your artifact deployed, like so:

```
`mvn firebase:run`
```

Please note that the 'firebase:run' target does not clean or build the project, so in reality you may want to run this instead:

```
`mvn clean package firebase:run`
```