

# **Cubeia Firebase**

## **Developer Reference**

**Lars J. Nilsson**

**Fredrik Johansson**

**Tobias Westerblom**

**Viktor Nordling**

**Peter Lundh**

---

# Cubeia Firebase: Developer Reference

Lars J. Nilsson

Fredrik Johansson

Tobias Westerblom

Viktor Nordling

Peter Lundh

Firestore Version 1.7.0-CE

Published 01/21/2010

Copyright © 2009, 2010 Cubeia Ltd

This work is licensed under the Creative Commons Attribution-Share Alike 2.5 Sweden License. To view a copy of this licence, visit <http://creativecommons.org/licenses/by-sa/2.5/se/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

---

---

---

# Table of Contents

I. Introduction .....	1
1. About Cubeia .....	2
Cubeia Ltd .....	2
Contact .....	2
2. System Overview .....	3
Introduction .....	3
Game and Services .....	3
Events and State .....	4
Game Object Space .....	4
System State .....	4
Message Bus .....	4
Replication and Failover .....	4
Game and Tournament Nodes .....	4
Client Nodes .....	8
Master Nodes .....	8
Transactions .....	8
Current Implementation .....	8
Known Limitations .....	8
Java Transaction API .....	9
Further Reading .....	10
Persistence .....	10
JDBC .....	10
JPA .....	13
Platform Domain Data .....	15
Lobby .....	15
Chat .....	15
Users and Clients .....	15
Connect .....	15
Login .....	16
Join / Play .....	17
Logout .....	17
Disconnects .....	17
Second Login .....	18
Class Loading .....	18
Hierarchy .....	18
Service Contracts .....	19
Shared Libraries .....	19
Public Services .....	19
DOS Protection .....	20
Plugin Services .....	21
LoginLocator .....	21
PlayerLookupService .....	21
LocalHandlerService .....	21
ChatFilter .....	22
SystemInfoQueryService .....	22
II. Server Development .....	24
3. Server Game Development .....	25
What's a Game? .....	25
Game Life Cycle .....	25
Sending Events .....	25
Scheduling Events .....	26

Event Processing .....	27
Handling the State .....	27
Game Processor .....	27
The Game Interfaces .....	27
Game .....	27
TableInterceptor / Provider .....	28
TableListener / Provider .....	28
TournamentGame .....	28
Game Activation .....	29
The GameActivator .....	29
The Default Activator .....	30
Packaging .....	31
Deployment .....	31
4. Tournament Development .....	32
What is a Tournament? .....	32
Tournament Life Cycle .....	32
Tournament Activation .....	32
The GameActivator .....	32
5. Service Development .....	34
What is a Service? .....	34
Contract and Implementation .....	34
Service Lifecycle .....	35
Configuration .....	36
The ServiceRegistry .....	36
Receiving and Sending Events .....	36
Packaging .....	37
Service Descriptor .....	37
Structure .....	38
Dependencies .....	38
Exporting Classes .....	38
External Libraries .....	39
Archive Access .....	39
Deployment .....	39
6. Lobby Development .....	40
7. Miscellaneous .....	41
Using the Cluster Configuration .....	41
Unified Archives .....	41
Exploded Deployment .....	42
Shared Class Loader .....	42
Services .....	42
Example UAR Content .....	42
Failure Detection .....	42
Configuration .....	43
Handling Pings In the Client .....	43
8. Building With Maven .....	44
Repository .....	44
Archetypes .....	44
Game Archetype .....	44
Service Archetype .....	44
Tournament Archetype .....	44
Project Archetype .....	45
Packaging .....	45
Build Plugin .....	45
Packaging Type .....	45

Example Game POM .....	45
Running Firebase .....	46
III. Client Development .....	48
9. Client Game Development .....	49
Network Protocol .....	49
Table Usage Scenarios .....	49
Joining Table .....	49
Reconnections .....	49
Creating Custom Tables .....	50
Table Invites .....	51
Connection Failover .....	51
Disconnection Handling .....	51
Reconnection Handling .....	51
Notify Watching .....	52
A Client Developer's Guide to Tournaments .....	52
Registering and Unregistering .....	52
Tournament Starting .....	52
Player Out .....	52
Player Moved .....	52
Tournament Finished .....	52
Lobby .....	52
Client API .....	53

---

## List of Figures

2.1. Replication Sequence Diagram .....	5
2.2. Replication with Transaction Sequence Diagram .....	6
2.3. User Transaction Sequence Diagram .....	7
2.4. Class Hierarchy Outline .....	18

---

## List of Examples

2.1. Failing User Transaction .....	7
-------------------------------------	---

---

# Part I. Introduction

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

---

# Chapter 1. About Cubeia

## Cubeia Ltd

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  
Stora Nygatan 33  
11127 Stockholm  
Sweden

Email: info (at the cubeia domain)

Corporate Homepage: <http://www.cubeia.com>

Community Community: <http://www.cubeia.org>

---

# Chapter 2. 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.

## 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 the services are static, local 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**      The client node handles client connections and session management.

**game**        Game nodes are responsible for game deployments and actions.

**master**     Master nodes control the cluster topology and manages communications.

**mtt**        Tournament nodes (Multi Table Tournament) controls tournaments.

Normally one node is loaded per server and kept running for the server lifetime. However, if 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. Nodes can be combined in any manner on a server but for stable systems one node per server is recommended.

A game is a module loaded by the game nodes 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.

## Game 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 always 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 or more times, and may keep multiple instances alive at the same time. Services on the other hand are always loaded as singletons.

## Events and State

A Firebase cluster balances traffic between nodes and provides transparent fail-over 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 Object Space

The Game Object Space (GoS) is a distributed cache which exists "between" the game nodes in order to share table data. Each table 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 servers in batches, where the GoS (see above) instantly replicates delta changes the system state will group changes into batches and replicate on a configurable interval.

## Message Bus

The message bus (MBus) 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.

## Replication and Failover

These sections describes the replication process and how the system handles server down if fail-over is enabled.

## Game and Tournament Nodes

This section describes the replication and fail-over (if enabled) 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 JavaSpace 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).

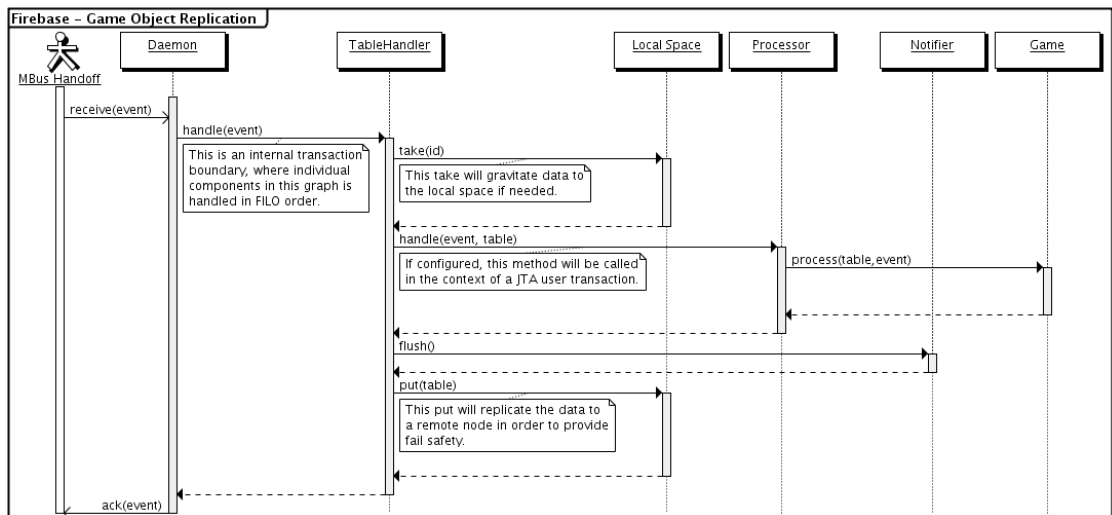
- get (int ID)** Will return an object from the space. Acquires a read lock.
- take (int ID)** Will return an object from the space. Acquires a write lock which is kept until the object is returned to the space via 'put'.
- put (object)** Returns an object to the space. This will be done with a write lock. If we already have a write lock (from 'take') we will release it.

## Replication

An object will be replicated between servers (i.e. space instances) 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 (by a 'put'). The replication can be configured in multiple ways, depending on the requirements by the user.

Below is a sequence diagram of executing an event for a table including the space calls.

**Figure 2.1. Replication Sequence Diagram**

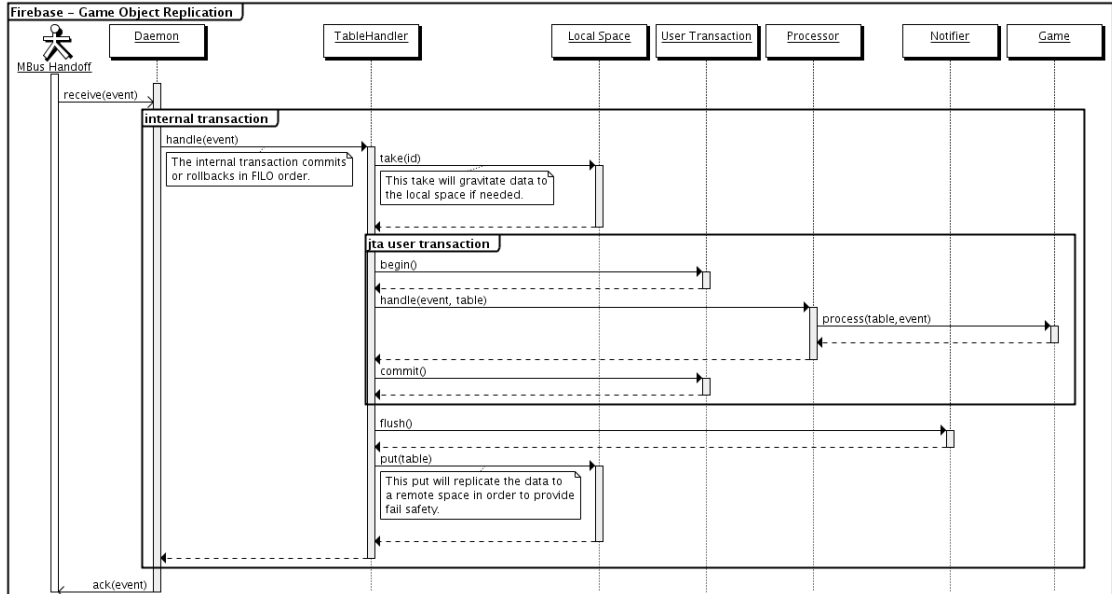


## Transactions

When accessing a game object (table or tournament) with a 'take'/'put' we will always access it within a transaction. Thus, the execution of events on game objects are also always performed within a transaction.

Below is the replication sequence again but with the transaction boundaries added.

**Figure 2.2. Replication with Transaction Sequence Diagram**

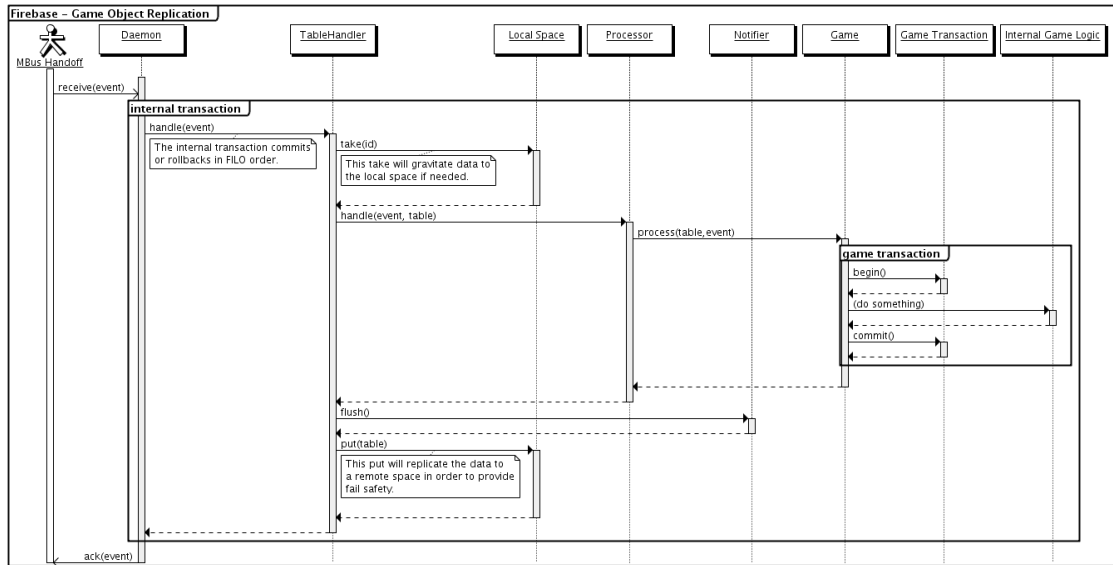


Firestore supports two different types of transactions to be used. The default mode is a local user transaction that will not be visible to the game implementation. The optional mode is using a JTA transaction manager which will provide an XA transaction that the game implementation can use. The JTA manager implementation used at the time of writing is Bitronix [<http://docs.codehaus.org/display/BTM/Home>].

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) although performance may suffer. This has nothing to do with the internal wirings of Firestore and everything to do with the fact that two phase XA transaction implementations are expensive.

## 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 handling like in the sequence below (some calls have been omitted for clarity).

**Figure 2.3. User Transaction Sequence Diagram**

For the scenario above it is important to realize that the nested transaction ('game transaction' in the sequence diagram) can very well be committed without the 'internal transaction' being committed properly. A roll-back of the 'internal transaction' will not cause a roll-back of the 'game transaction'. In this scenario it is therefore important to write robust handling for roll-back scenarios for the 'game transaction'.

If 'game transaction' fails and is rolled back, you can throw a `RuntimeException` to roll back the 'internal transaction'. This cannot, however, be applied the other way around.

### Example 2.1. Failing User Transaction

Let's say you are executing a user event ('event1') regarding a monetary transaction from an account to the given table. You make the database writings within a user transaction and commits, then something goes awry and an exception is thrown which will result in the 'internal transaction' getting rolled back. The state of the table have now been rolled back and will not reflect the state-changes you started to make while executing event1, but the storage facility (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 probably recommended regardless application) so that the money is not moved again.

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 the example. A good way of solving the unknown variable is letting the service manage duplicate calls (i.e. idempotency). Failing to do so in a service implementation will for most systems lead to unwanted behaviour.

## Failover

When a server goes down (with 'goes down' we mean a hard, non-recovering crash such as power failure or kill -9 on the process; software related crashes such as out-of-memory, infinite-loops or thread-blocking will most likely not be detectable as a server down by the system), the system will detect the loss of a member and assign another server with the same node type to take over.

For example, if we have 4 servers, 2 client and master + 2 game, then if one game node crashes we will execute all tables on the remaining game node.

The most complex behaviour when a fail over occurs is when the crash will terminate a thread within event execution. As stated in the transactional section, if you have already committed your own user transactions then they are committed and will not be rolled back, reverted etc.

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 re-iterate, when a game 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 (using the message bus/internal ESB) is not transactional and outgoing messages (i.e. packets to clients) 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 for server to server messaging. To make a robust fail over solution this also needs to be addressed in the application.

The guarantees are:

- When a server goes down the state of objects on that server will be restored from the last committed event (i.e. fully executed).
- Scheduled game actions are restored but will be rescheduled with the initial delay.
- There will not be any duplicate executions of committed events.

## Client Nodes

[TBW]

## Master Nodes

[TBW]

## Transactions

Firestore uses a JTA transaction over each processed event. The transaction is used to optimise network performance for state transfers 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.

## Current Implementation

Firestore utilises a JTA implementation from Bitronix [<http://docs.codehaus.org/display/BTM>]

## Known Limitations

Database and JDBC driver support for XA is somewhat lacking. Please make sure your choice of database supports XA properly before integrating. Firestore 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 [<http://www.bitronix.be/Btm/DatabasesXASupportEvaluation>].

Firebase does not support XA recovery.

Firebase does not support distributed XA transactions.

## Java Transaction API

### Integrations

#### XA Data Sources

Data sources can be configured as XA data sources in which case they automatically will be part of the event processing `UserTransaction`. 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>
```

#### JTA Persistence

Configuring the persistence manager to use the JTA `UserTransaction` 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>system-xa</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>
```

```
</properties>
</persistence-unit>
</persistence>
```

## 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 is:

- The JTA Specification [<http://java.sun.com/products/jta/>]
- The XA Specification [<http://www.opengroup.org/pubs/catalog/c193.htm>]
- XA Exposed, pt. I, II and III [<http://jroller.com/page/pyrasun?catname=%2FXA>]

## Persistence

Firestore supports two ways of communicating with a database, plain JDBC and JPA. Both can be configured to be used in a XA transaction context or using local transactions.

## JDBC

Firestore 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 are setup by Firestore, and all connections will be pooled for performance.

## Deployment and Configuration

To deploy a datasource, specify the connection properties in a `<name>-ds.xml` and place it in the deployment folder of the server.

A simple data source config file may look like this:

```
<?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">root</entry>
  <entry key="password"></entry>
</properties>
```

If the datasource file above is contained in a file with the name 'foo-ds.xml', then you can reference the datasource in Firestore by the name 'foo'.

## Data Source Transactions

The `tx-type` property specifies the data source transaction type. This can be used to setup XA data sources. Available values are:

**NON-TX** Non-XA data source, vanilla JDBC connection pooling.

**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.

## Properties

### Global

**tx-type** Data source transaction type - [ NON-TX | LOCAL-TX ] - default: NON-TX - Optional

### 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>
```

### LOCAL-TX

**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

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' period.

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.

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 LOCAL-TX

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 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' period.

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>
```

## JPA

Firestore enables Java Persistence archives to be deployed similarly to JDBC data sources. It uses Hibernate [<https://www.hibernate.org/>] as the underlying JPA implementation.

You can deploy a persistence archive as a stand alone persistence archive ('.par') or bundled with the game ('.gar'). Using a separate persistence archive ('.par') allows all resources to access the persistence archive. Using a persistence archive inside a game archive ('.gar') scopes the persistence archive to that specific game only.

Reference documentation for Hibernate can be found here [[http://www.hibernate.org/hib\\_docs/entitymanager/reference/en/html](http://www.hibernate.org/hib_docs/entitymanager/reference/en/html)].

Below is some example code for persisting a `TestItem` within a game, using a persistence context named 'test':

```
GameContext context; // This will be injected in 'init' method

private void createTestItem() {
    // Get the persistence service from the service registry
    ServiceRegistry reg = context.getServices();
    Class key = PublicPersistenceService.class;
    PublicPersistenceService service = reg.getServiceInstance(key)
    // Get the entity manager for the persistence unit named 'test'
    EntityManager em = service.getEntityManager("test");
    // Create the bean
    TestItem item = new TestItem();
    item.setText("some data");
    persist(em, item);
}

private void persist(EntityManager em, TestItem unit) {
    // Omitted for now, see below
}
```

The method for performing the `persist` call will be different depending on if you are using local transactions or XA transactions. This can be specified in the `datasource` and in the 'persistence.xml'.

**Persisting using a non-JTA data source.** When using a data source which does not support JTA you need to begin and commit a local transaction yourself. It is important that you always release the transaction after executing.

```
protected void persist(EntityManager em, TestItem unit) {
    EntityTransaction transaction = em.getTransaction();
    try {
        transaction.begin();
        em.persist(unit);
        transaction.commit();
    } catch (Exception e) {
```

```

        transaction.rollback();
    }
}

```

**Persisting using a JTA data source.** If you are using a JTA data source you will be using XA transactions. If you are executing inside a game there will already be a transaction declared and you do not need to begin a transaction.

```

protected void persist(EntityManager em, TestItem unit) {
    em.persist(unit);
}

```

## Deployment and Configuration

A persistence archive follows the specification of Java EE 5. The persistence archive configures a persistence unit and contains persistence entity beans and can be deployed in the server deployment folder.

The content of the persistence archive should match...

```

/<classes>
/META-INF/persistence.xml

```

... where <classes> indicates that the compiled Java classes should be included (and not in a separate JAR file). Below is an example of a 'persistence.xml' file (formatted for space reasons):

```

<?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="RESOURCE_LOCAL">
    <non-jta-data-source>system</non-jta-data-source>
    <class>com.cubeia.firebase.api.entity.User</class>
    <properties>
      <property
        name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5Dialect" />
      <property
        name="hibernate.hbm2ddl.auto"
        value="update" />
    </properties>
  </persistence-unit>
</persistence>

```

The deployed persistence unit will use the deployed data source with name as specified in the data source tag (that is, <jta-data-source> <non-jta-data-source>). So for the example 'persistence.xml' above, you should also deploy a valid data source for the name 'system', in other words, a data source configured in a file named 'system-ds.xml'.

The persistence deployment will scan the persistence archive for entity beans and bind them all to the persistence unit automatically. If you want you can also explicitly specify any entity class in classpath that you want to bind to the persistence unit (in the example, the `User` class). Using explicit binding can be useful if you want to bind entities not contained in the archive.

It is also possible to deploy a persistence unit within a game archive ('.gar'). To do this, simply include a 'META-INF/persistence.xml' final in you game archive and it will be deployed. However, this persistence unit will be scoped within the game's class loader so it will not be possible to use it from another context since the persistence entity beans will not be reachable for other class loaders.

## Platform Domain Data

A game object in Firebase handles only events for tables ("areas"), 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.

### Lobby

All tables 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 table in the lobby has a path, for example. `/realMoney/10plyrs/table21`, and a set of attributes describing the table. The platform will automatically arrange the lobby, but also has options for customising the lobby path and the attributes of a table when the table is created.

The structure of the lobby as static and defined when a game is developed. Client and server developers will have to agree on a lobby model to use, the server game developers populate the lobby and make sure changes are properly propagated, and the client developers uses a subscription model with delta changes to manage the lobby representation in the actual game clients.

### Chat

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. As such, it is trivial for the developers to add, remove and use the chat system.

## 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 user name and password.
- join/play**    Join tables and play games.
- logout**      End client session and close communication channel.

### Connect

You connect to firebase by opening a TCP socket on a server running a gateway node and using the port specified in the server configuration file 'cluster.props'. The default port is 4123. The server will not send out any handshake packet or expect any specific first packet from the client (this is likely to change in the future).

The client can issue the following platform specific commands before being logged in:

<b>login</b>	Authenticating user with user name and password.
<b>lobbyQuery</b>	Retrieve and query the lobby data.
<b>version</b>	Exchange protocol version numbers with the server.
<b>gameVersion</b>	Exchange a specific game version number with the server.

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 will not be safe to continue.

## Login

A client can send a login request. The packet is defined with the following fields: '

<b>user</b>	String
<b>password</b>	String
<b>operator ID</b>	int
<b>credentials</b>	byte array

The credentials is a byte array with arbitrary data which can be used by the `LoginHandler`. The `LoginHandler` is service which can be deployed to handle authentication for an installation.

## 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 a service with the contract `LoginLocator`. For example:

```
import com.cubeia.firebase.api.action.local.LoginRequestAction;
import com.cubeia.firebase.api.login.LoginHandler;
import com.cubeia.firebase.api.login.LoginLocator;
import com.cubeia.firebase.api.service.Service;

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 one of those as well. The responsibility of the `LoginHandler` is to authenticate the user, then verify or deny the login request.

```
import java.util.concurrent.atomic.AtomicInteger;
import com.cubeia.firebase.api.action.local.LoginRequestAction;
import com.cubeia.firebase.api.action.local.LoginResponseAction;
import com.cubeia.firebase.api.login.LoginHandler;

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, please have a look at the service section of this manual for those details. We don't need to configure Firebase as this is a so called 'plugin service', meaning that Firebase will use it automatically if it is deployed.

## Join / Play

The client use system `JoinTableRequest` to join tables. After which game play is defined by the games themselves. The system will keep track of which tables each client is watching and which tables they are seated at. If a client joins or watches a table, a reference for that client will be set in the system state.

## Logout

Then the server receives a logout command it will clean up resources for the client on the server. This includes:

- 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.

## 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 time period (configurable in server configuration for 'cluster.props' as the property 'node.client.client-reconnect-timeout'), a reaper will remove the client from the system state.

If the client reconnects before the reaper time, then he will receive notifications for each table watching and seated at. Lobby and filtered join requests will not be resumed so if you want a transparent fail-over you need to re-subscribe in your client.

If the client reconnects after the reaper time, then the reference to seated tables etc. have been cleaned up and he will not receive any notifications.

## Second Login

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.

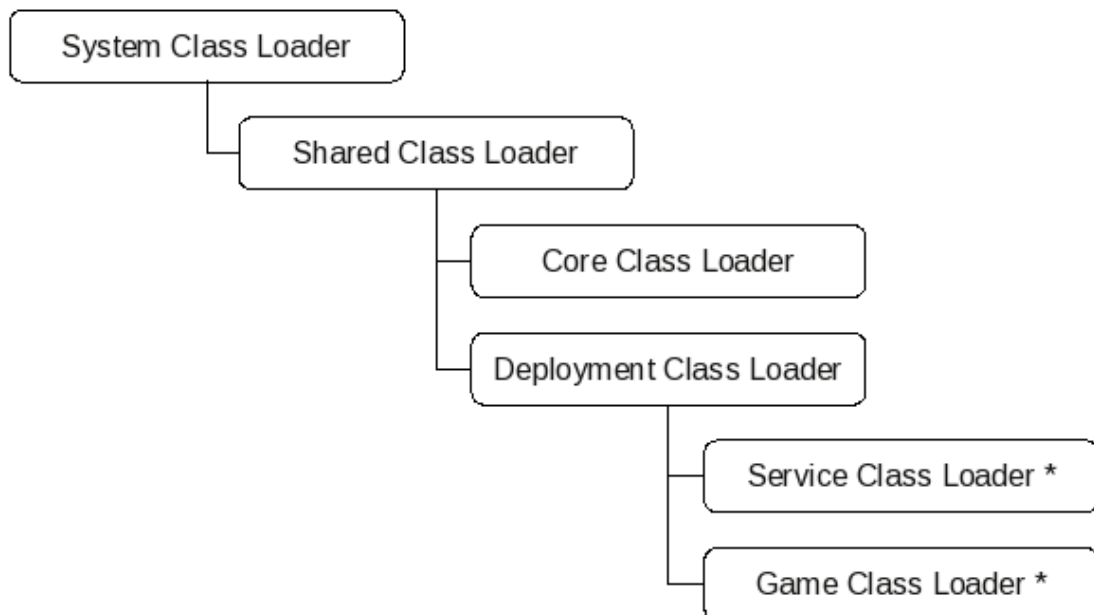
## Class Loading

Like many Java Enterprise servers Firebase keeps an advanced class loading hierarchy in place to support code isolation and make hot deployment of code units at all possible. Each game and service will be loaded in its own private class loader. Games communicate only by the defined game API. Services can export classes, in effect making them usable in the entire server.

## Hierarchy

Class loaders are formed in a hierarchy use parent / child relations ships. Below is a flattened tree of the class loader types involved in a Firebase installation. A '\*' at a class loader marks that the class loader occurs mutiple times, for example there will be one class loader for each game.

**Figure 2.4. Class Hierarchy Outline**



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.

<b>system</b>	Includes 'bin/bootstrap.jar' and 'lib/common/*.jar'.
<b>shared</b>	Contains a dynamic export table for system wide classes.
<b>core</b>	Internal archives and services, sourced from 'lib/internal/*'.
<b>deployment</b>	Shared classes for deployed artifacts, from 'game/lib/*.jar'.
<b>service</b>	Sourced from individual service artifacts, 'game/deploy/*.sar'.
<b>game</b>	Sourced from individual service artifacts, 'game/deploy/*.gar'.

From this follows that all JAR files placed in './lib/common' will be shared by both servers and games, and the JAR files in './game/lib' will be universal for the deployed games and services. Also, that no JAR or SAR files in './lib/internal' will be available to the deployed games.

## Service Contracts

Services are deployed and made available across the entire server. This is done via the method called 'exporting', where the service contract is automatically exported by virtue of extending `Contract`.

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'. This is because service contracts are loaded by the 'shared' class loader in order to be accessible through the entire server.

## 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 and/or services, place them in './game/lib'.
- To share libraries across services, either have one 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 re-deployment, and will be universal for the entire server.

## Public 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`, a reference to the service would be obtained like this:

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

**ClusterConfigProviderContract.** 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 (see below) and one which is configured at the master node and propagated across the cluster.

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

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

**PublicPersistenceService.** This service provides access methods for all deployed JPA entity managers. Deployment of JPA archives is described in the persistence deployment section.

**PublicSystemService.** 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.

**SystemTransactionProvider.** 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.

**PublicClientRegistryService.** This is the publicly available client registry which manages all connected clients and sessions.

## DOS 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. You can lookup the service by using the contract like this:

```
serviceRegistry.getServiceInstance(DosProtector.class);
```

## How-To

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 "\_" is 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
```

```
}
```

Please refer to the Java API documentation for more information.

## Plugin Services

Firebase has a number of services which are defined as 'plugin' services. These are typically used to extend core server behaviour, if a specific plugin service is not found, Firebase reverts to its default behaviour.

## LoginLocator

The login locator extends Firebase to provide authentication for an installation. Please refer to the 'Writing a Login Handler' section for details.

## PlayerLookupService

Player lookup is connected to the player query request/response available to the clients. The player query response contains a byte array which can be used for implementation specific data. To be able to take advantage of this you need to implement a specific `PlayerLookupService`.

A player query 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.

To provide you own lookup service, implement and deploy a service that implements the `PlayerLookupService` interface.

## LocalHandlerService

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.

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);  
    }  
}
```

```
}  
}
```

## ChatFilter

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.

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 step 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.

Please refer to the Java API documentation for more information on the `ChatFilter`.

## SystemInfoQueryService

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 3rd parties to amend extra information to the response. To be able to take advantage of this you need to implement a specific `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 p = new Parameter<Integer>("Jackpot", next, Type.INT);  
        action.getParameters().add();  
    }  
}
```

```
    return action;  
}  
[...]  
}
```

---

# Part II. Server Development

This part of the reference manual deals with the server side development of games, services and tournaments.

---

# Chapter 3. Server Game Development

This section discusses Firebase games, their roles in a Firebase cluster and how to develop and deploy games on a Firebase server.

## What's a Game?

A game in Firebase is a class which implements the `Game` interface and can be deployed in a Firebase server. Is defined as the business logic for a multiplayer game. A game is localised on a table and it is implemented to handle only as many players as there are seats available.

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.

A model approach of games, tables and state would be:

```
[Game Logic] ---> [Table] ---> [Game State]
```

The game logic will be 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.

## Game Life Cycle

A game has a life cycle which is managed by the platform. It is important not to make assumptions about a particular game instance. The platform may start/stop objects at any time. It may keep several instances alive at the same time (so called object pooling) and it may start the games lazily.

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 only 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.

## Sending Events

The `GameNotifier` object is provided to the games for purposes of sending events to clients connected to the Firebase server. The `GameNotifier` can be fetched from the table by calling `Table.getNotifier()`. Note that the `GameNotifier` should not be kept as a reference that might be serialized.

The `GameNotifier` 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`. The `GameEvent` will usually be of the type `GameDataAction`. The `GameDataAction` has a method called `setData(ByteBuffer)`, which is used for appending the actual event. The `ByteBuffer` should contain the data that the client should receive. Firebase puts no restriction on which protocol to use when creating the `ByteBuffer`. One could, although it is not recommended, use a plain string as a protocol and just write it to a `ByteBuffer`. More efficiently, some binary protocol should of course be defined for the game.

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

```
import java.io.IOException;
import java.nio.ByteBuffer;

import com.cubeia.firebaseio.action.GameDataAction;
import com.cubeia.firebaseio.action.GameObjectAction;
import com.cubeia.firebaseio.game.GameNotifier;
import com.cubeia.firebaseio.game.GameProcessor;
import com.cubeia.firebaseio.game.table.Table;

public class MyProcessor implements GameProcessor {

    [...]

    private void sendHelo(Table table) throws IOException {
        GameNotifier notifier = table.getNotifier();
        // Get UTF-8 bytes and wrap
        byte[] bytes = "Hello 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);
    }
}
```

## Scheduling Events

Firebase support loopback scheduling by providing a `TableScheduler` which is available via the table, through the `Table.getScheduler()` method. Scheduled game actions are delivered back to the game after a given delay in milliseconds.

The `TableScheduler.scheduleAction(GameAction, long)` method is used to schedule an action, and returns a `UUID` which acts as an identifier for the action. Please note that this is a loopback actions, which will be inserted into the `Game` when scheduled, it will not be sent to any client. It is typically used for scheduling a timeout when a player should act.

The `TableScheduler.cancelScheduledAction(UUID)` can be used to cancel an already scheduled action using the ID returned when the action was scheduled. This is typically done when an action is scheduled for a timeout for a player, but the player did act in time.

It should be noted here, that Firebase cannot guarantee that the timeout does not occur at the same time as the player's action is being handled, in which case the scheduled action cannot be cancelled. Therefore, the game must still verify that a timeout that occurs is expected. This can be done by checking the identifier of the scheduled action when it is received by the game.

## Event Processing

[TBW]

## Handling the State

[TBW]

## Game Processor

The `GameProcessor` is the key interface for a game. The implementor of this interface is the point of entry for all actions occurring in the game. Two methods are described:

- `handle(GameDataAction, Table)`
- `handle(GameObjectAction, Table)`

The first method is the one that will be most frequently called. More exactly, every action that a player performs in a game will generate one call to this method. The `Table` contains the game state and can be fetched by calling `Table.getGameState()`. To handle the action, get the data from the `GameDataAction` by calling `GameDataAction.getData()`. This will return a `ByteBuffer` containing the game specific data for this action. This allows for the game to specify its own protocol for game packets. It is then up to the game to parse the data in the `ByteBuffer` and then handle the action accordingly.

In order to respond to an action, the `GameNotifier` should be used. For more information about the `GameNotifier`, please see the section on sending events.

The second method, which takes a `GameObjectAction` is generally used for internal actions. An internal action is typically something that is scheduled by the game. The reason for the different type of action is that an internal action will probably never be sent to clients, so it is more convenient to be able to attach the action data as an `Object`, rather than a `ByteBuffer` that a `GameDataAction` takes as data. For example, if the game asks a player to act and the player should time out in 10 seconds, the game can schedule a `GameObjectAction`, with a `TimeoutAction` as data, to be executed in 10 seconds. This will result in the `handle(GameObjectAction, Table)` method being called 10 seconds later, with the `TimeoutAction` contained in the `GameObjectAction`.

## 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 [...] {  
  
    /**  
     * @param con Game context, never null  
     * @throws SystemException If the game fails to initialize  
     */  
    public void init(GameContext con) throws SystemException;  
  
    /**  
     * 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();  
  
    /**  
     * Called when the game will no longer be used.  
     */  
    public void destroy();  
  
}
```

## TableInterceptor / Provider

The `TableInterceptor` interface can be implemented by the game in order to control seat, leave and reservation requests on the table, which is 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 `TableInterceptor` can return false when the `TableInterceptor.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.

If the game does not want to implement `TableInterceptor` directly, it can instead implement the `TableInterceptorProvider` which only contains a getter for a `TableInterceptor`.

## TableListener / Provider

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.

If the game does not want to implement `TableListener` directly, it can instead implement the `TableListenerProvider` which only contains a getter for a `TableListener`.

## TournamentGame

If a game participates in a tournament, this interface should be implemented. It provides a tournament processor used to start and stop tournament 'rounds'. Please refer to the tournament documentation for more details.

# Game Activation

Game activation occurs when a game is deployed. It is controlled by a `GameActivator` class which is invoked by Firebase to handle initial table creation and general system setup. The activator to use for a specific game is configured in the game deployment descriptor.

## The GameActivator

A game is configured with a game activator in the deployment descriptor using the "activator" element. 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. That is: 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. More on this below.

`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 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 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.

## 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 (see above) 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

[TBW]

## Deployment

A game is deployed in a game archive. The game archive is defined as a ZIP file with the extension "gar" (Game Archive). The content of the game archive should match:

```
/*_jar
/GAME-INF/game.xml
/GAME-INF/lib/*_jar
```

The class implementing the `Game` interface should be placed in a JAR in the root of the GAR file. JAR files placed in the root or in the lib ('/' or '/GAME-INF/lib') will be loaded by the class loader. However, only JAR files in the root will be scanned for annotations that may be used for persistence services for instance.

---

# Chapter 4. Tournament Development

This section briefly discusses Firebase tournaments, their relation to games and how to develop and deploy tournaments on a Firebase server.

## What is a Tournament?

A game in Firebase is a class which implements the `MTTLogic` interface by sub-classing the `MTTSupport` class. 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 more than one tables and are commonly referred to as MTT (Multi Table Tournaments). A tournament that is played out on a single table is commonly referred to as a STT (Single Table Tournament). STT's can be implemented in the regular game logic and does not have to use the firebase tournament framework.

## 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 (so called object pooling) 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.

## Tournament Activation

Tournament activation occurs when a tournament is deployed. It is controlled by a `TournamentActivator` class which is invoked by Firebase to handle initial tournament creation and general system setup. The activator to use for a specific tournament is configured in the tournament deployment descriptor.

## The GameActivator

[TBW]

## 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 file name, 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.

---

# Chapter 5. Service Development

This section discusses Firebase services, their roles in a Firebase cluster and how to develop and deploy service on a Firebase server.

## What is a Service?

A service in Firebase is a module consisting of at least two classes, one which extends the `Contract` interface and one which implements the contract extension and also implements the `Service` interface. It is a software module, 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 service stack, and it's services are treated as first class members in a server, and is a prerequisite without which the server will refuse to start. If any service reports an error at start-up the server will halt immediately.

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 also implement their own services to provide common functionality between the games.

A service is accessed via a service registry. The registry is a simple map of services where each service is associated with a public id (a globally unique string), as well as their contract (more on the contract below). The service contracts are separated from their control methods and will only cooperate with each other and the games via their public 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 service which is not installed as a core Firebase service is also '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.

## Contract and Implementation

A service is logically described by two classes, its contract interface and its concrete implementation class. 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 details 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 which implements the public interface and the `Service` interface.

An example contract may look like this:

```
import com.cubeia.firebase.api.service.Contract;

public interface HelloWorldService extends Contract {

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

}
```

The above contract extends `Contract` and details the services 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:

```
import com.cubeia.firebase.api.server.SystemException;
import com.cubeia.firebase.api.service.Service;
import com.cubeia.firebase.api.service.ServiceContext;

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. The `Service` 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. For example, if service `X` implements service contract interface `A` doing a reverse cast such as `(X = (X) A)` may fail with a class cast exception.

## Service Lifecycle

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

```
public interface Service [...] {

    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 initiate 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 initiate method because it will not get another chance to halt the execution of the server.

The service interface is not re-entrant. 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 initiated. However, services are lazily started only when accessed the first time.

## Configuration

A service has three main methods of configuration:

1. Through the Firebase server configuration, as described here.
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, as described in the deployment section. 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 in the service archive and then use property files in the server configuration directory to override the default values.

## The ServiceRegistry

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

The group 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 are uniform across a cluster of several servers. In other words, all services should be deployed on all servers. This may seem excessive for services which may not, due to deployment reasons, be used on certain servers, but as the services are lazily started this should be safe.

Services are accessed from the registry via their contract interface or via their public id. The by far most used method takes the contract interface and attempts to return an instance of the given contract. Should the contract not be found, null is returned. For example:

```
ServierRegistry 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 formatted file which describes the service the archive contains. Below is an example service descriptor, containing all legal elements:

```
<service auto-start="false">  
    <name>A Simple Example Service</name>  
    <public-id>ns://www.cubeia.com/examples/service</public-id>  
    <contract>com.cubeia.example.service.SimpleService</contract>  
    <service>com.cubeia.example.service.impl.ServiceImpl</service>  
    <description>This is a simple example service.</description>  
    <dependencies />  
    <exported>  
        <package>com.cubeia.example.service.*</package>  
    </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 be a readable title for the service. The public ID is a globally unique string identifier, normally modelled on a name space, 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 normally 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 detail 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 reference 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

A service is deployed in a service (SAR) archive. The archive is defined as a ZIP file with the extension "sar". The content of the service archive should match:

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

JAR files placed in the root or in the lib ('/' or '/META-INF/lib') will be loaded by the class loader, with the exception of exported classes (please refer to this page for class loading issues).

---

# Chapter 6. Lobby Development

[TBW]

---

# Chapter 7. 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 `ClusterConfigProviderContract` 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 your namespace and configuration interface. The method names are matched to properties much like an ordinary Java beans matching, but they are also prefixed by the namespace. Some examples might be:

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

4. You'll access the configuration via the `ClusterConfigProviderContract` `getConfiguration(Class, Namespace)` method. The namespace for a configuration interface is normally set in the `Configured` 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)

## 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.

## 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 for failure detection. In short the server will determine if a client is deemed 'idle' and start pinging it, and one or more missed ping responses will disconnect the client.

If ping is enabled, given client session 'X':

- For each received packet within X update time stamp X.y
- If  $(X.y + (\text{initial-ping-delay})) > (\text{system-time})$ , start pinging
- For each ping:
  - Wait <ping-timeout>
  - If ping timeout:
    - If  $(\text{ping-no}) \geq (\text{failure-threshold})$ , 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 'service.ping.ping-enabled' to 'true'.

A client will be deemed 'idle' if the server does not get any data from it within 'service.ping.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 'service.ping.ping-interval' in milliseconds.

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

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

## Handling Pings In the Client

The Firebase API's 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.

---

# Chapter 8. Building With Maven

## Repository

In order to utilise the Cubeia Firebase Maven tools, you need the following repository added to you 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 and TAR projects. There's currently no archetype for multi-module projects or UAR projects.

### Game Archetype

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

```
mvn archetype:generate \
  -DarchetypeGroupId=com.cubeia.tools \
  -DarchetypeArtifactId=firebase-game-archetype \
  -DarchetypeVersion=1.7
```

### Service Archetype

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

```
mvn archetype:generate \
  -DarchetypeGroupId=com.cubeia.tools \
  -DarchetypeArtifactId=firebase-service-archetype \
  -DarchetypeVersion=1.7
```

### Tournament Archetype

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

```
mvn archetype:generate \
  -DarchetypeGroupId=com.cubeia.tools \
  -DarchetypeArtifactId=firebase-tournament-archetype \
  -DarchetypeVersion=1.7
```

## Project Archetype

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

```
mvn archetype:generate \  
  -DarchetypeGroupId=com.cubeia.tools \  
  -DarchetypeArtifactId=firebase-project-archetype \  
  -DarchetypeVersion=1.7
```

## Packaging

The Firebase archive plugin handles packaging for Firebase games, service tournament and unified archives. The normal Maven packaging rules apply, with the exception of the archive deployment descriptors, which should be placed in a sub-folder 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 (GAR): /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 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.tools</groupId>  
      <artifactId>archive-plugin</artifactId>  
      <version>1.7</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>1.7.0-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.tools</groupId>
      <artifactId>archive-plugin</artifactId>
      <version>1.7</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>1.7.0-CE</version>
  <configuration>
    <deleteOnExit>false</deleteOnExit>
  </configuration>
</plugin>
```

[...]

The 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
```

---

# Part III. Client Development

This part of the reference manual deals with the client side development of games and the Firebase APIs.

---

# Chapter 9. Client Game Development

This section briefly discusses Firebase client development and the Firebase client API's.

## Network Protocol

[TBW]

## Table Usage Scenarios

This section describes common scenarios when joining and leaving tables.

### Joining Table

A client can join a table either by:

- Join directly, by sending a `JoinRequestPacket`.
- Send a watch and then subsequently a join.

Which flow you should use is usually dictated by the game design. For example, in most poker clients you open a table first (watch) and then select a seat (join).

The communication flow differs for the two cases. When a watch is sent Firebase will respond with seat info packets for all players currently at the table. If you then join the table, then no seat info packets will be sent (they are regarded redundant). However, should you join a table without watching first then you will receive a seat info packet for each player.

### Reconnections

If the client for some reason drops the connection or crashes and is restarted, then when the user re-logs in (manually or automatically) then Firebase will notify the client what tables he/she was seated at and/or watching.

### Seated Tables

At all tables the user was seated at, the player-status has been set to `WAIT_RECONNECT`. The player object has not been removed. When the user re-logs in he will receive a stream of notify seated packets, one for each table he/she was seated at. If the client sends a join request to the tables, the user will be take the original place and the player status will be set to `CONNECTED`. If the client does not acknowledge the seat in a set amount of time, the player will be set to status `DISCONNECTED` and the client will not be prompted for the seat again if he reconnects a second time.

### Watched Tables

For all tables the client was watching Firebase will send a notify watching packet. The watcher association will be kept by the server until the client sends out an unwatch table or the reaper removes the session after a disconnect timeout.

### Timeout

If the client loses connection and the timeout configured for the client reaper occurs then all session information regarding the client will be removed from the system. If you reconnect your client after a

reaper has timed you out it will be like a new connection, you will not receive any notifications of what tables you were seated at and/or watching.

The player object at the tables will not be forcibly removed by the server, but flagged as disconnected. It is up to the game implementation what to do with disconnected clients. Therefore it might be possible to see reaped clients still seated and (seemingly) participating in games.

## Reserved Seat

The waiting list will reserve a seat before notifying the player. The reservation procedure seats a player with the correct player id in the seat with status set to `RESERVATION`. This is in order to avoid notifying players of seats that are taken and there's a client race conditions. If the client does not acknowledge the reservation notification within a set amount of time we will silently remove the player. The possible actions will generate callbacks (server side) as follows:

- Seat reserved -> `TableInterceptor.allowJoin(...)` and `TableListener.seatReserved(...)`
- Seat taken -> `TableInterceptor.allowJoin(...)` and `TableListener.playerJoined(...)`
- Seat forsaken (timeout) -> Nothing
- Seat rejected -> `TableInterceptor.allowLeave(...)` and `TableListener.playerLeft(...)`

The handling regarding seat reservation is similar to reconnections.

## Creating Custom Tables

Clients can request tables to be created and automatically receive a reservation on the new table. They can also, in the same step, send invitations to other player who will also get reserved seats at the new table. This is done by sending a `CreateTableRequest` from the client to the server. This request will be handled by the server together with the game activator and a new table will eventually be created. The server will answer with a `CreateTableResponse` which will have its status set to '0' if the table have been created and the seat id reserved for the player, the client should claim the reserved seat using an ordinary join request.

A `CreateTableRequest` may optionally contain a list of invitee ids. The ID's will be matched to players and these players will receive a `NotifyInvited` action informing them that there is a seat reserved for them at a table. Again, if a client receives this event, it should answer with an ordinary join request, otherwise the seat reservation may lapse.

In order for this to work, the game activator, on the server, must implement the `RequestAwareActivator` interface. This interface allows the activator to react on incoming requests and participate when the table is created. This is similar to ordinary table creation but also contains options for modifying the list of invitees and determining if seats should be reserved for invitees. The activator may also reject the request. If the activator does not implement the `RequestAwareActivator` interface an error will be printed in the logs.

## Reserved Seats

The creator and optionally the invitees will get seats reserved at the table. This will happen automatically when when the table is created. However, the `TableListener` will not be informed of the reservation.

The upshot of which is that if the creator or the invitees disconnect before that have claimed the seats, using a standard join request, the `TableListener` will get a status change to `WAITING_RECONNECT` just as if this were an ordinary reservation but without first having been explicitly told about it.

## Table Invites

Clients may be invited to games, in which case a `NotifyInvited` will be received. The client should use a join request to claim the reserved seat. Clients may invite other clients by sending a `InvitePlayersRequest`. The server may simply reserve a seat and send out `NotifyInvited` to the affected clients.

## Connection Failover

In the event of the client losing the connection with the server you need to reestablish a new connection.

## Disconnection Handling

In the server, the following actions are taken when a client disconnects (without an explicit logout):

- The distributed session status is set to `DISCONNECTED`.
- Unwatch on all known watching tables.
- Update all known tables to player status `WAIT_REJOIN`

In the event of a disconnect right after a join request to a table, we might have a boundary case where the player status on the table is still `CONNECTED`.

## Reconnection Handling

In the client, when reconnecting and logging in you will receive the following:

- `NotifySeated` packets, one for each table you were seated at.
- `NotifyWatching` packets, one for each table you were watching.

In the event of a reconnect directly after a join request was sent, you might be able to receive a join response for that table. The reason for this is that messaging is asynchronous within the system, and if you have successfully reconnected before the response is delivered to the gateway tier, then we will deliver it to you.

In the event of a reconnect after the response was discarded but before the player/table association state has propagated (for the newly joined table), you may have a player seated at a table with the status `CONNECTED`. The client session will pick this up when the table is communicating and send a `notify seated` for that table. There will only be one `NotifySeated` per table.

For example:

1. A client send a join request to table A.
2. The client disconnects and reconnects to another client node after the join response was handled but before the state has propagated.
3. Now the table will have a player seated and the client node was not able to tell the client that he was seated there since the state has not yet reached the client node. In the case above the client will receive the `NotifySeated` when the table tries to communicate with the client.

## Notify Watching

This packet means that you were watching the provided table in your previous session. You should either acknowledge this by sending a watch request or unwatch request. If you send a watch request you will start to receive table events. If you send an unwatch request the association will be removed from your session and should you reconnect again you will not receive a notify watching. If you do not acknowledge the notification then the association will still be present in your session until either you acknowledge it or the reaper cleans up your session (timeout or logout).

## A Client Developer's Guide to Tournaments

Here we will describe the key things to think about when implementing support for tournaments on the client side.

### Registering and Unregistering

To register to a tournament, send a `MTTRegisterRequest` packet for the tournament ID you want to register to. You will receive an `MTTRequestResponse` from the server. If the registration was successful, the response status will be OK. Note that if the same client registers to the same tournament again, the server will return response status OK again.

### Tournament Starting

When the tournament starts, players registered for the tournament will receive an `MTTSeated` packet. At this point, the player will be registered as a seated player at a table in the tournament and will thus be sent packets for everything that happens in the tournament. The client must therefore make sure to react on the `MTTSeated` packet by opening up a table where the actions can be shown.

### Player Out

When a player is out of the tournament, he will receive an `MTTPickedUp` packet. The packet's 'keep\_watching' flag will be set to true. This means that the player can keep watching the action at the table where he was seated. It is up to the client to decide when to stop watching the table, by sending an `UnwatchRequest` packet.

### Player Moved

When a player is moved to a new table, he will receive an `MTTPickedUp` packet. The packet's 'keep\_watching' flag will be set to false. This means that the client should close (or clear) the current table. As with the `MTTSeated` packet, the player will automatically receive all subsequent actions on the new table.

### Tournament Finished

There is currently no Firebase packet for notifying a client that the tournament is finished. This is intentional, because it is up to the tournament implementation to define the logic and send messages to the client as appropriate.

## Lobby

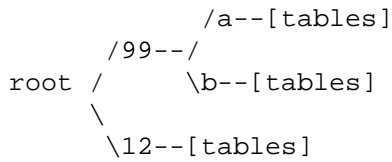
Firestore exposes all tables in the system through a lobby. The lobby can provide the user with lobby data by two means.

- Lobby query
- Lobby subscription

A lobby query will always return full table packets. Lobby subscription will use a more fine-grained update model where you will receive a full update first and then you will receive delta updates to the first set of table packets. Using a subscription model lowers server CPU usage and bandwidth usage drastically.

The lobby in Firebase is arranged according to a tree. The only fixed structure is that the first node-level after the root must be the game id.

For example:



In the lobby tree above, we have two games deployed, game id 99 and 12, respectively. The game with game id 12 does not have any branches in its tree, but the game with id 99 has two branches, 'a' and 'b'.

Let's say a table with table id 4 exists for the game with id 99 and resides in the branch a. The lobby path would then be: /99/a/4. See the Java API documentation for the class `LobbyPath` for more information about the lobby path.

## Client API

[TBW]