Cubeia Back Office – Web Services Overview

Version: 1.2

Contents

Scope

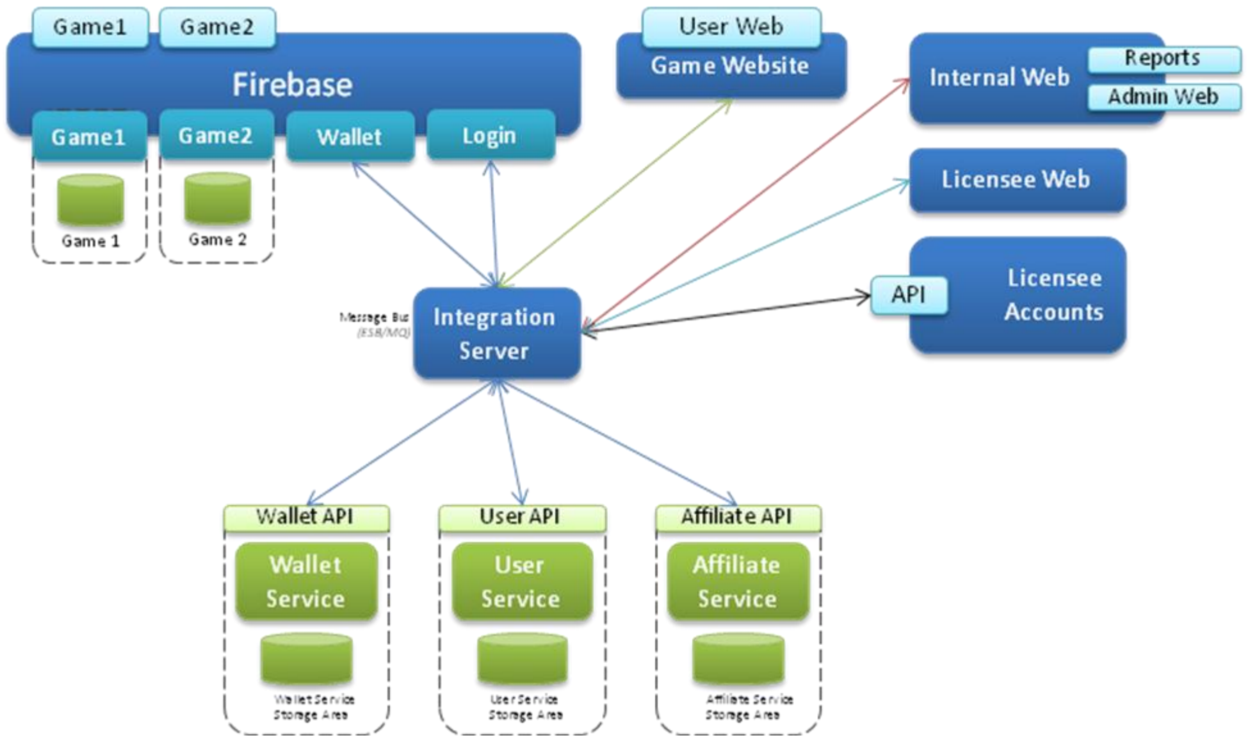
The aim of this document is to describe the common elements of the web services that are part of Cubeia Network. These common elements include but are not limited to messaging and error handling.

Cubeia Network Overview

Cubeia Network is modeled after SOA/WOA practices. The different services are exposed through a specified protocol to clients that can send in synchronous requests directly, or in some cases; asynchronous requests via a message queue. The network components are all isolated and stand alone. It is possible to deploy and use them separately or together.

Both point to point and hub and spoke integrations are supported. But the hub and spoke topology is the recommended as this limits the complexity.

Below is an example of a hub and spoke topology setup using Cubeia Back Office web services for user and wallet solution.



Using a web/service oriented architecture means that the system will be modularized with low coupling and high cohesion.

Integration server

The integration server can consist of a service bus (ESB) with the main responsibility to act as a router of messages between the services or a message queue MQ.

Using a service bus/MQ and a hub and spoke topology is preferred to using point to point integrations. Hub and spoke means that the integrated services only needs to talk directly to the broker which will route, and if needed transform, the call to the destination service. It also enables the deployment of service spanning business processes to be developed and deployed in the integration server.

An ESB can help when integrating with legacy systems, a payment service provider for example, as it supports message transformation and a large amount of transport protocols (JMS, file, FTP, HTTP, etc).

Using an MQ provides a lightweight alternative to an ESB and is good for systems that wants include a centralized broker and take advantage of asynchronous messaging (EDA/SEDA).

Point-to-point topology provides a simpler setup and is a good starting point for smaller system.

Communication

Request / Response

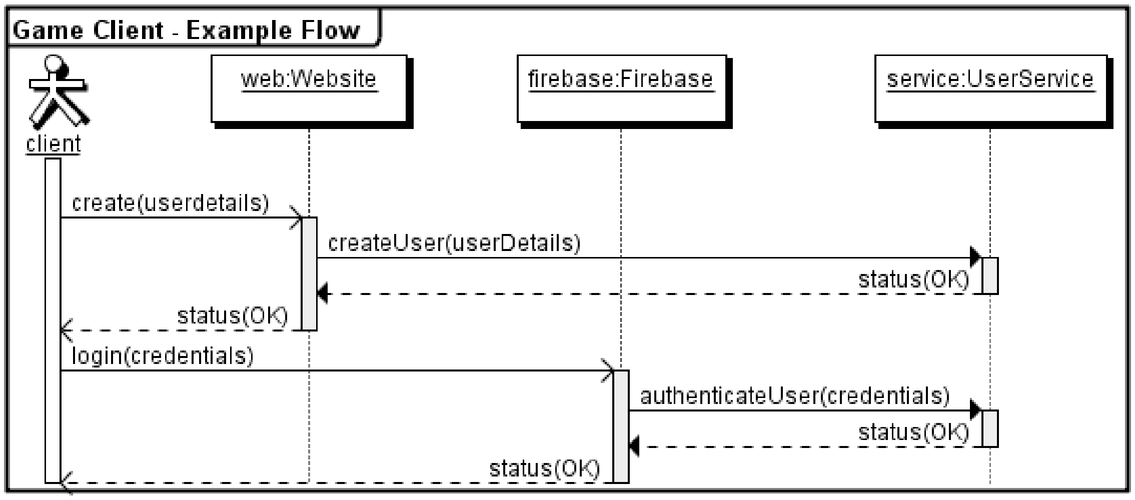
All synchronous communications with the services are initiated on the client side (a client is anyone trying to communicate with the service). The client will send a request and the service will answer with the specified response.

Some requests are defined as transactional and some are defined as read-only. Please read the section regarding transactional requests for more information.

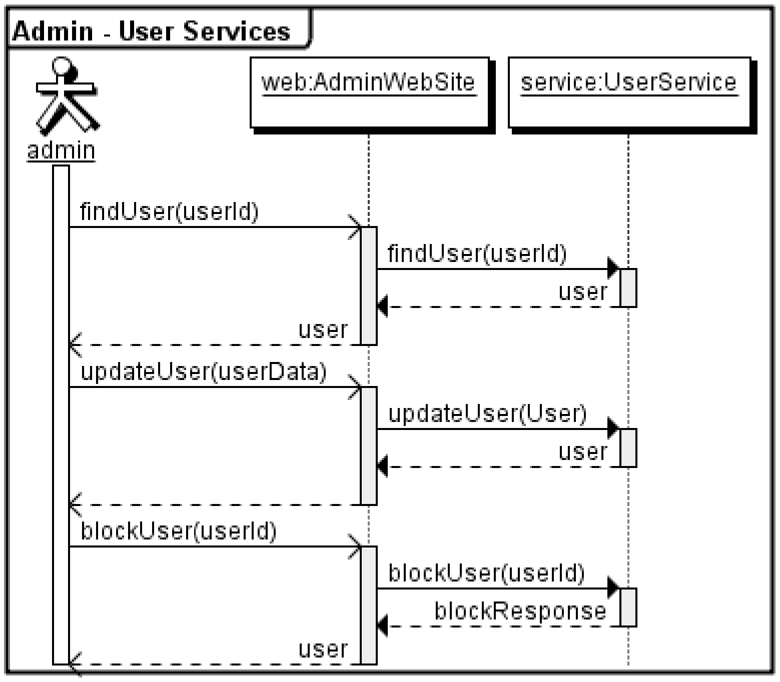
Basic Communication Flow

Below are some examples of a communication flows for a client that calls a service through some intermediate layer(s).

Example of create user request and then login through the game client and game server:



Example of an administrator using the User Service through an admin web frontend:



Idempotence

All HTTP methods except POST are considered idempotent by design. This means that you can safely retry a GET on a resource within changing the internals.

For POST type resources you risk of running modifications when you retry the same request.

Some resources therefore have a transactional id which will be picked up and used by the service layer to provide an idempotent API even for transactional POST requests.

Handling Failure

Requests to the service are distributed calls and as such they are prone to failure. If a client A sends a request to service S and does not receive a response a couple of things may have happened.

Let's simplify the service's request broker method:

```
Response DoRequest(Request) {
    ProcessMessage(Request);
    Return response;
}
```

We need to acknowledge that failure can occur at any given time. A failure can happen:

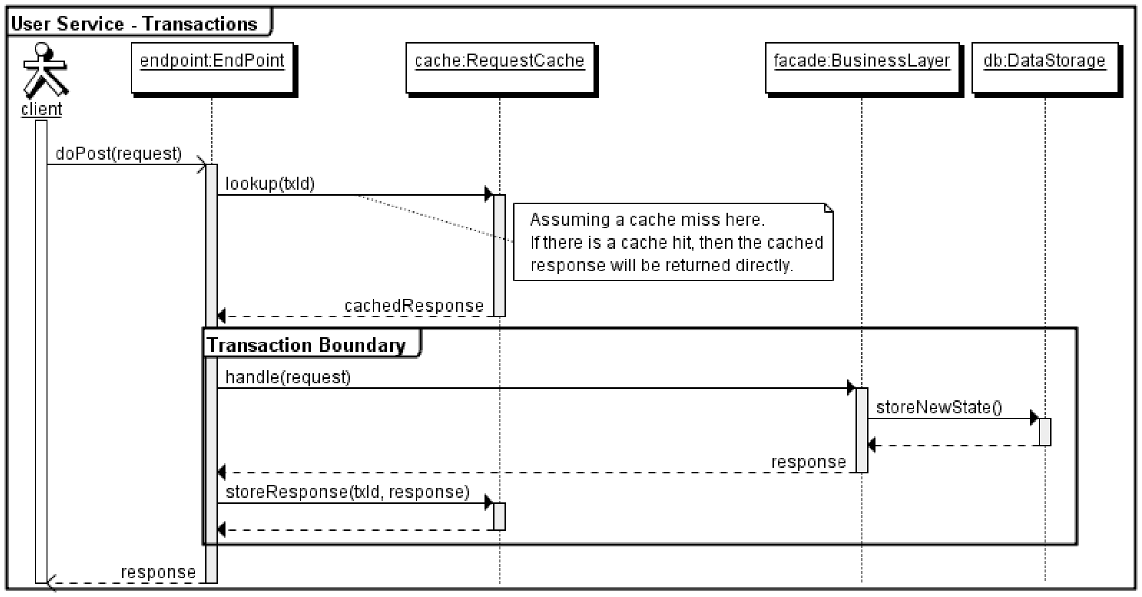
1. Before the message has been processed,
2. During the processing of message,
3. After the processing of message but before returning the response.

To address the different scenarios, the user service is idempotent. For any given transactional request, we will always return the same response but never executing the same logic again.

What this means is practice is that it is safe to apply a retry pattern from the client.

If the business logic was not allowed to process completely then nothing will be written to the database. If the business logic was allowed to processed completely then we will have a cached response that will be used to provide an idempotent response if the request is received again.

The internal transactional boundaries are shown in the picture below.



Request messages that do not contain a transaction id (txId) field, are either idempotent by nature or are not deemed critical to re-execute.

If we go back to the original problem where A sends a request to S and does not receive a response, what should A do?

The correct approach would be to retry the request *n* times with a timed delay of at least a second (where n should not be a very high number). If the transaction is still not responded by the server, then A should mark it as failed, store it in a log and finally notify any administrative service that the transaction might need manual attention.

If a request is found in the cache then the parameters are checked against the incoming request. If the parameters do not match then an Error Response is sent back to the client.

Error Handling

HTTP Status Codes

The API always attempts to return appropriate HTTP status codes for every request.

- **200 OK**: Success!
- **304 Not Modified**: There was no new data to return.
- **400 Bad Request**: The request was invalid. An accompanying error message will explain why. This is the status code will be returned during rate limiting.
- **401 Unauthorized**: Authentication credentials were missing or incorrect.
- **403 Forbidden**: The request is understood, but it has been refused. An accompanying error message will explain why. This code is used when requests are being denied due to update limits.
- **404 Not Found**: The URI requested is invalid or the resource requested, such as a user, does not exists.
- **406 Not Acceptable**: Returned by the Search API when an invalid format is specified in the request.
- **500 Internal Server Error**: Something is broken. Check the logs for error messages.

- **503 Service Unavailable**: The service is down. This can be for various reasons.

Error Messages

The API will always attempt to include an error response to the applicable error status code (404 not included). The content of the returned error message is specific to each component and you need to consult the component's protocol specification.