

Walteri Hyvönen

# Custom Mobile Game Analytics Implementation

Bachelor's thesis  
Information Technology / Game Programming

2018



**Kaakkois-Suomen  
ammattikorkeakoulu**

<b>Tekijä/Tekijät</b>	<b>Tutkinto</b>	<b>Aika</b>
Waltteri Hyvönen	Insinööri (AMK)	Joulukuu 2018
<b>Opinnäytetyön nimi</b>		40 sivua 2 liitesivua
Kustomoidun pelianalytiikkaratkaisun toteutus mobiilille		
<b>Toimeksiantaja</b>		
Nitro Games Oyj		
<b>Ohjaaja</b>		
Niina Mässeli		
<b>Tiivistelmä</b>		
<p>Tämä opinnäytetyö on dokumentti kustomoidun palvelinpuolen pelianalytiikkasysteemin toteutuksesta ja sen vaiheista. Analytiikkaratkaisun tarkoituksena oli luoda helposti muokattavissa ja jatkettavissa oleva pohjarakenne ilmaismoninpeliin nimeltä Medals of War, joka korvasi nykyisen kolmannen osapuolen liitännäisen.</p> <p>Tämän dokumentin on samalla tarkoitus toimia oppaana geneerisen tapahtumapohjaisen analytiikkaratkaisun toteutukselle. Asianmukaisen ymmärryksen takaamiseksi tärkeimmät vaiheet on käsitelty syvällisesti sekä pohjattu funktionaalisiin vaatimuksiin.</p> <p>Opinnäytetyötä lähestyttiin toteutuspainotteisesti. Google BigQueryn ja Singularin yhteiskäyttö palvelinpuolen analytiikkaratkaisun kanssa oli kaikille osallisille uusi aihe. Koska aiheesta oli vain vähän edeltävää tietoa, suuri osa toteutuksesta koostui kokeilemisesta ja epäonnistumisesta. Opinnäytetyön edetessä monia verkkoviestinnän ja asynkronisen prosessoinnin aiheita tutkittiin ja hyödynnettiin.</p> <p>Käyttäjän puolen koodin toteutukseen käytettiin Unity-pelimoottoria, palvelinpuolen toteutus rakennettiin olemassaolevan Node.js-palvelimen päälle. Tapahtumien rakenne toteutettiin ohjelmistovaatimusten ja datan määrittelyjen mukaisesti. Erityisesti huomiota kiinnitettiin geneerisen datanlähetyspotken kehittämiseen, sen ollessa koko järjestelmän ydinominaisuus. Käyttäjän ja palvelimen väliseen viestintään käytettiin HTTP REST -ohjelmointirajapintaa, ja samalla käytettiin hyväksi projektin valmiita ominaisuuksia.</p> <p>Työn aikana kehitetty toimiva geneerinen pelianalytiikkaratkaisu, joka soveltuu hyvin jatkokehitykseen. Analytiikkaratkaisu on helposti tuotavissa muihin projekteihin sekä opinnäytetyön loppuvaiheessa joitain ominaisuuksia tuotiin jo muihin projekteihin.</p>		
<b>Asiasanat</b>		
Unity, pelianalytiikka, BigQuery, verkkotyöskentely, Singular, Google, massadata, Node.js		

<b>Author (authors)</b>	<b>Degree</b>	<b>Time</b>
Waltteri Hyvönen	Bachelor of Engineering	December 2018
<b>Thesis title</b>		40 pages
Custom mobile game analytics implementation		2 pages of appendices
<b>Commissioned by</b>		
Nitro Games Plc		
<b>Supervisor</b>		
Niina Mässeli		
<b>Abstract</b>		
<p>This thesis studies the implementation of a highly customisable event-based analytics framework for the mobile free-to-play online multiplayer game, Medals of War. The objective of this thesis was to provide a modifiable and extendable analytics system that could replace the current third-party plugin. This thesis was additionally intended to act as a guide for future implementations. In order to ensure proper understanding of the implementation process, crucial steps are examined in depth and based on the functional requirements.</p>		
<p>Developing a custom analytics framework in conjunction with Google BigQuery and Singular was a new topic for everyone involved in this thesis study. As the general knowledge of this topic was limited, development consisted of a great amount of trial and error. During development, various topics such as networking and asynchronous processing were studied extensively.</p>		
<p>Client-side development was conducted using a game development platform called Unity, while the backend part of the analytics system was built as part of a Node.js server. Special attention was given to the implementation of a generic event sending pipeline as it is the core feature of this custom analytics system. Communication between clients and the server was accomplished by utilizing a HTTP REST programming interface with pre-existing features.</p>		
<p>The finished product was a functional, extendable and generic game analytics system that can easily be implemented to other projects. The thesis study was deemed a success and provides an excellent basis for future development.</p>		
<b>Keywords</b>		
Unity, game analytics, BigQuery, networking, Singular, Google, big data, Node.js		

# SISÄLLYS

1	INTRODUCTION .....	7
1.1	Purpose of this thesis .....	7
1.2	Initial approach .....	7
1.3	Nitro Games Plc.....	8
1.4	Medals of War.....	8
2	TOOLS AND BACKGROUND INFORMATION .....	9
2.1	Event-based analytics.....	9
2.2	BigQuery.....	9
2.3	Singular .....	10
2.4	Programming languages.....	10
3	FUNCTIONAL REQUIREMENTS .....	11
3.1	Customisable events .....	11
3.2	Analytics event placement .....	12
3.3	Mandatory analytics data .....	12
3.4	Data validation .....	13
4	IMPLEMENTATION.....	14
4.1	BigQuery project setup .....	14
4.1.1	Creating a project .....	14
4.1.2	API Credentials.....	17
4.2	Backend implementation .....	18
4.2.1	Authentication .....	18
4.2.2	Creating an endpoint .....	19
4.2.3	Processing event data .....	22
4.2.4	Singular events .....	27
4.2.5	Server event placement .....	29
4.2.6	Event testing .....	30
4.3	Frontend implementation .....	32

4.3.1	Singular SDK .....	32
4.3.2	Event queue.....	33
5	CONCLUSIONS .....	36
	REFERENCES .....	38
	FIGURES.....	40
	ATTACHMENTS.....	41

## ABBREVIATIONS

IAP	In-app purchase
API	Application Programming Interface
MoW	Medals of War
GCP	Google Cloud Platform
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
SDK	Software Development Kit
URL	Uniform Resource Locator

## **1 INTRODUCTION**

Moving into the era of mobile games where the so-called freemium business model reigns supreme, the importance of data analysis and player profiling is massive. These freemium games offer in-app purchases for players that grant some form of virtual goods, for example additional in-game resources. In-app purchases and advertisements are the main sources of income for a large amount of mobile games, especially ones with social aspects. Being able to analyse and deduct popular features and in-app purchases is the key to better retention and income rates.

### **1.1 Purpose of this thesis**

This thesis aims to provide information on how to implement event-based custom game analytics for a mobile game using BigQuery. The game analytics implementation is done for a free-to-play mobile real-time strategy game called Medals of War that is developed and published by Nitro Games Plc.

This thesis examines the theory behind the implementation process and reasons for the selected approaches and methods used. The purpose is to describe the process of creating an easy-to-modify analytics system in the form of a comprehensive guide. The study should be easy to follow and its contents applicable to other similar projects if needed.

The study introduces the tools used along with some necessary background information and theoretical framework. Also, the requirements of this analytics system and explanations for the selection of tools and methods are presented. The main focus of this thesis is on the implementation itself where the process is explained step by step. Finally, the whole process is reviewed to identify what still needs to be done and which aspects could be improved.

### **1.2 Initial approach**

With no previous experience or knowledge of backend programming, the implementation was approached carefully. A lot of time went into studying and asking questions from more experienced colleagues, but most of the implementation process consisted of experimenting with trial and error.

One of the key goals was to eventually remove the old analytics system and replace it with a better and more customisable one. A similar analytics implementation already existed at the time of starting this thesis which served as a basis for this implementation. Many parts of the base analytics framework had to be changed to match the requirements of this thesis. Ways to implement features and properties are compared when possible, and decisions are made after considering various approaches.

### **1.3 Nitro Games Plc**

Nitro Games Plc is a Finnish mobile game developer and publisher. Nitro Games was founded in 2007 and their headquarters is in Kotka, Finland. Nitro Games converted to a publicly listed company on Nasdaq First North Stockholm on the 16th of June 2017. (Nitro Games Plc 2018.)

The company focuses on developing free-to-play competitive multiplayer games for mobile devices, even though it started out as a PC-game development company. The goal of this thesis is to improve the online strategy game Medals of War by developing a more flexible game analytics framework. (Nitro Games Plc 2018.)

### **1.4 Medals of War**

Medals of War is a real-time competitive multiplayer strategy game developed and published by Nitro Games. The game follows the freemium business model. All content in the game can be accessed for free, but the players are offered additional resources in the form of IAP's. (Nitro Games Plc 2018.)

Medals of War is a card collecting game where players construct battle decks to use in battle against other players in a World War 2 fantasy setting. Battle decks are built around Officers with special abilities, using various units and commands. There are many social features included such as companies, live-events, challenges and leaderboards. Players can compete in a solo and/or group environment in the effort to climb the rankings. (Nitro Games Plc 2018.)



## **2 TOOLS AND BACKGROUND INFORMATION**

This chapter contains some necessary background information and information about the tools used for the implementation. As the subject of this thesis is very broad, the theory in this chapter only portrays the knowledge required for this specific project.

### **2.1 Event-based analytics**

Event-based analytics is a data tracking method based on user behaviour. Events are recorded data of an instance of user behaviour and are tied to that user's identity. The most important aspects are to identify who did something, what they did and under what circumstances it occurred. Based on context and continuously gathering data linked to a user, different kinds of conclusions can be drawn. Event data can be used to track user progress in a certain sequence, for example a game tutorial or a shopping cart checkout. (Egan 2016.)

Event-based analytics is at the core of any modern business in the gaming industry. Massive amounts of gathered data are irreplaceable in improving user experience and revenues through structured analysis. The power of gathering data from events is that it provides the possibility for real-time analysis and a specific context for each piece of data. (RoboMQ n.d.) The environment and minds of customers in the gaming industry are known to be volatile and ever changing. Event-based analytics provides an effective solution for recording and reacting to any changes in player behaviour. It enables the business to predict emerging trends, minimise risks and enforce popular features. (Datamatics n.d.)

### **2.2 BigQuery**

Google BigQuery is a serverless data warehouse that offers highly scalable storage and data analysis. Data can be either sent as batches or streamed through powerful streaming ingestion. Analysis can be extended with various advanced machine learning algorithms if deemed necessary to obtain even more information out of gathered data. BigQuery's fast streaming insertion API

enables real-time analytics that is extremely helpful for event-based analytics in games. (Google 2018.)

Data is queried using standard SQL, and usage is priced per amount of data scanned. Flexible pricing also affects other Google Cloud services, thus the user is only charged for what they use. Permissions can be set with data and role-based configurations through Cloud IAM to keep data on a need-to-know basis. (Google 2018.) Google additionally offers extensive documentation on how to implement these services into a project in various programming languages to help with initial implementation.

### **2.3 Singular**

Singular is a popular marketing intelligence and analytics platform used to record marketing related data such as purchases made by users. The platform includes a fraud detection system to quickly spot suspicious user behavior and react to it. Most importantly for this thesis, Singular supports custom in-app events and metrics. Receipts from all IAP's are sent to the Singular dashboard for analysis via their HTTP REST API. (Singular 2018.)

Singular enables player grouping and profiling for targeted content offers. The dashboard offers various utilities and visualizations for gathered data, including custom metrics and group tagging. (Singular 2018.) For this thesis, the use of Singular is limited to communicating with their Web API. The Singular dashboard and metrics are not covered as they are beyond the scope of this thesis.

### **2.4 Programming languages**

Two main programming languages are used in the implementation of the analytics system, C# for frontend code and Node.js for backend code. These languages were not chosen for any specific reason apart from the fact that the game has already been written with C# and Node.js. Due to Node.js being based on JavaScript, which is fairly similar to C#, having limited previous knowledge was not a large problem and learning was quick.

The game client uses Unity3D as its game engine. Unity supports C# and UnityScript, a JavaScript-like language, but started the deprecation process for UnityScript in 2017 (Fine 2017). In addition to Unity deprecating the other available programming language, C# is simply more widely used. This means finding other programmers with knowledge of the language and the ability to help with specific problems is much more likely. Node.js is a widely used language for server-side programming as well. It is an open-source, cross-platform environment used to run JavaScript on a server machine.

Traditionally, JavaScript is run on a client browser, but Node.js makes it possible to process content before being sent to the user's client. From a real-time multiplayer game point of view, Node.js offers great capabilities for asynchronous operations necessary for real-time communication. (Orsini 2013.)

### **3 FUNCTIONAL REQUIREMENTS**

Analytics implementations are heavily tied to the projects they are made for and their requirements change drastically depending on various needs. Each project requires its own type of analytics solution and for MoW, a combination of client and server triggered events was found appropriate.

It is important to keep in mind that an analytics system might never be complete as it can be extended almost endlessly. Due to this, there will be no strict specifications for the final product. It is still necessary to have general functional requirements since they will guide the implementation in a right direction. This chapter examines the functional requirements for MoW and provides a general description of the key elements in event-based game analytics.

#### **3.1 Customisable events**

For an event-based analytics system, being able to customise event structure and content is immensely important. The structure of customisable events is defined with a schema that includes information of the data included. Schemas contain fields of data, each with a specified name and data type. Event and field names are used later to query data for analysis so they must be set as clearly and descriptively as possible.

For this thesis, all events are sent under the same table in the event dataset. The schema for this table contains all common data as separate fields and one JSON field for additional event parameters. This field can contain any data as their own data fields and, therefore, can be modified without needing to change the schema itself. Without having to change the structure, changes to the sent data can be made on the fly. It can be difficult to analyse additional parameters as they need to be extracted separately resulting in slower performance. BigQuery charges customers based on the amount of scanned data, causing costs to increase as well. (Cooladata 2017.) In this thesis, a third-party company will be fetching data from BigQuery and processing it for analysis and visualising the data through a customised application.

### **3.2 Analytics event placement**

This analytics implementation for MoW consists mainly of server triggered analytics events with a few exceptions. Most events are sent when the server recognises that something happened, the rest are requested by the client. Previously, all events were triggered and sent directly to the analytics platform from clients, which made it easier for skillful players to modify the sent data. In order to prevent receiving incorrect analytics data, as many events are set up server-side as possible. Sending events from the server minimizes the possibility of lost data due to client crashes as well.

Correct event placement is mandatory for data analytics. Incorrect event placement will result in skewed or completely false data which causes problems when doing analysis. Before implementing an event, its location must be considered carefully. First, where the event is placed, client-side or server-side, and reasons to do so must be considered. Every event that can be placed in the server code should be placed there, especially since in most cases the necessary data is already being processed. Sending an additional request to send an event is simply pointless if the same result can be achieved without it.

### **3.3 Mandatory analytics data**

Some information must be included in every single event that is sent to BigQuery. Mandatory data includes fields that can be used to identify the

player, specific event and device the player is using. All events must be tied to a player ID and have a unique event ID, so that all data belongs to someone and every event has a unique identifier. Additionally, an event timestamp and some data about the player and client device are required. For deeper analysis on player behaviour and grouping, factors such as player progression and currency, operating system, location and device model are tracked. This kind of data can be used to e.g. target special offers for players using an iOS or Android device.

Mandatory analytics data is updated every time a player launches the game by sending a network request to the server. The server should then create or update a database document for that player with the required data received in the request. Player progression data should not be saved under this document, but instead fetched separately as it often changes. The database document should include data that is needed for every event and does not frequently change. By doing this, the data does not need to be provided by the client all the time, and fully server-side events are made possible.

### **3.4 Data validation**

When doing data analysis, it is of utmost importance that the data is correct and reliable. Relevant and meaningful analysis cannot be done if the gathered data is incomplete or corrupted. In a worst-case scenario, analysing bad data can have a severe impact on the company itself. Data validation is a vital step in data analytics and ensures that all sent data is clean and useful. (McCulloch 2016.)

Due to this game analytics implementation being heavily server-sided, there is very little need for additional data validation. Nearly all data that is sent to the BigQuery data warehouse is fetched directly from the database, leaving the responsibility for correct data to the one implementing events. Event structures are configured in code, and data is validated using these configurations whenever something is sent. All unnecessary data fields are stripped from the event data, and structures that do not match a specific configuration are rejected.

## **4 IMPLEMENTATION**

Developing an analytics system may be considered an endless project. There is always something to improve on and more data to track. The implementation described in this chapter is not by any means complete and has much room for expansion. It should be noted that analytics solutions are heavily tied to a particular project and each implementation should be viewed as such. This chapter is divided into various subchapters that describe different areas of the implementation. Subchapters are presented in the order of implementation and can be used as a guide for future applications.

All events should be sent to the analytics platform from the server but triggering some of them on the server can prove to be impossible. When an event must be triggered on the client, it is not sent directly to the analytics platform. Instead, the event data is constructed into a JSON object and added to a queue. An event queue then takes up to a set maximum amount of event objects, combines them into a single bundle that is sent to the server as the body of a POST-request.

### **4.1 BigQuery project setup**

In order to start off the analytics implementation, a BigQuery project must be set up. BigQuery projects are managed on the Google Cloud Platform (GCP) website. A google sign-in is required to access the console and the services it has to offer. BigQuery is the analytics platform and data warehouse that is responsible for storing the analytics data and providing means to query that data. For this thesis, other data management services, for example machine learning algorithms, are not required. The BigQuery project, datasets and tables are the basic building blocks of BigQuery analytics required to store and query data. The following subchapters describe the process of creating these necessary elements.

#### **4.1.1 Creating a project**

The first step is to sign in to GCP and create the BigQuery project. The project is given a name and an organization to which it belongs. It must be noted that the given name is not the same as the project ID which cannot be changed

later as shown in Figure 1. The creator will be given the role of owner, granting access to everything under that project.

**Project Name \***  
My Thesis Project ?

Project ID: my-thesis-project-218307. It cannot be changed later. [EDIT](#)

**Location \***  
No organization [BROWSE](#)

Parent organization or folder

[CREATE](#) [CANCEL](#)

Figure 1 Creating the BigQuery project

A project has now been created, but to be able to store any actual data in the BigQuery data warehouse, datasets and tables need to be created. Datasets are data containers used to organise data and control access to the tables. Datasets can be created by selecting the project and clicking “Create Dataset” shown in Figure 2. During creation, a dataset is given a name, geographic location that cannot be changed later and the default table lifetime as seen in Figure 3. When querying data, all referenced tables must belong to datasets located in the same place. For this reason, all datasets in this project are co-located.

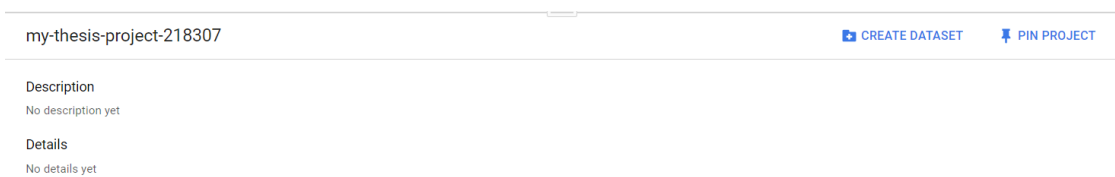


Figure 2 BigQuery project view

## Create dataset

Dataset ID

thesis\_dataset

Data location (Optional) ?

Default

Default table expiration ?

Never

Number of days:

Figure 3 Creating a BigQuery dataset

Now the project and the first dataset are ready, but a table is still needed to contain the actual data. The maximum number of tables a dataset can hold is unlimited, but close to 50,000 tables will cause slower performance (Google 2018.) There should be no worries of ever exceeding, or even coming close to this number as events are not stored in their own tables but in environment specific tables. Tables must be defined a schema, the data structure of that table. Schemas contain all wanted field names and their data types, and inserting data containing fields that do not exist in the schema will result in an error. Fields and their types in the schema are defined when creating a table but can be edited later if necessary. As shown in Figure 4, the table is also given a name, destination project and dataset. For development purposes, a table named “development” is created under the “events” dataset. Figure 4 shows declaring fields in the schema, which for this project include data such as user ID, a unique event ID, timestamp and name of the event and a field for event specific properties.



## Create table

## Source data

Create table from:

Empty table ▼

## Destination table

Destination project

Medals of War ▼

Destination dataset

events ▼

Table type

Native Table ▼

Destination table

development

## Schema

 Edit as text

Name	Type	Mode	
userID	STRING ▼	NULLABLE ▼	×
eventUUID	STRING ▼	NULLABLE ▼	×
clientVersion	STRING ▼	NULLABLE ▼	×
platform	STRING ▼	NULLABLE ▼	×
eventTimestamp	STRING ▼	NULLABLE ▼	×
eventName	STRING ▼	NULLABLE ▼	×
eventProperties	STRING ▼	NULLABLE ▼	×
+ Add field			

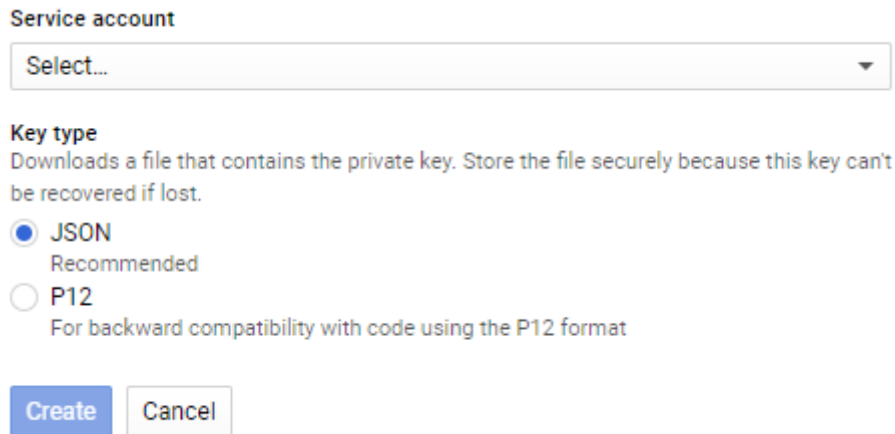
Figure 4 Creating a BigQuery data table

#### 4.1.2 API Credentials

In order to insert any data into the created tables, authentication is required. Authentication can be done via a Google service account that is specifically created to access Google API's. A service account is a Google account tied to the application itself rather than a real user, and its identity is used to authenticate the application. Permissions for a service account are set in the same way as any other Google account with preferably the minimum required settings. (Google 2018.)

Service accounts can be created from the IAM & admin section of the GCP. Once the service account has been given a name and role, a service account key must be downloaded as seen in Figure 5. This service account key holds

the credentials of the service account used to authenticate the application. The key can be downloaded as a JSON file which is then saved in a location that is accessible to the application.



The screenshot shows a web interface for downloading a service account key. At the top, there is a label 'Service account' above a dropdown menu with the text 'Select...' and a downward arrow. Below this is the 'Key type' section, which includes a warning: 'Downloads a file that contains the private key. Store the file securely because this key can't be recovered if lost.' There are two radio button options: 'JSON' (which is selected and has the word 'Recommended' below it) and 'P12' (with the text 'For backward compatibility with code using the P12 format' below it). At the bottom of the form are two buttons: a blue 'Create' button and a white 'Cancel' button with a grey border.

Figure 5 Downloading a service account key

## 4.2 Backend implementation

The core of this analytics implementation and the focus of this thesis is the backend implementation. As mentioned before, there are multiple reasons why it is desirable to have the system built server-side. When writing code for a system that is heavily influenced by current circumstances, it is important to remember that the environment changes. In an environment like this, code must be kept generic and extendable to avoid having to rework the whole system when something unexpected happens. In order to keep the code neat and organised, every event processed by the server follows the same single pipeline. Each event has their own configuration which is used to apply needed operations during processing.

### 4.2.1 Authentication

As previously mentioned, Google BigQuery requires authentication to access their API's. The server uses the service account key JSON file as credentials for authentication. Singular, the marketing intelligence platform, also requires authentication. As opposed to BigQuery, Singular does not need more than an API key on the server, which can be acquired from the Singular dashboard.

The server is authenticated by reading the server environment variable `GOOGLE_APPLICATION_CREDENTIALS` that should contain the file path of the downloaded service account key. It should be noted that this environment variable is only applied to the current session. Because it is reset if a new shell session is opened, setting the environment variable is tied to launching the server. During testing on a Windows machine, the command `$env:GOOGLE_APPLICATION_CREDENTIALS="[PATH]"` is run for the current shell session. The actual server, on the other hand, is a Linux machine which means the variable can be set in the server start-up script using `Environment=GOOGLE_APPLICATION_CREDENTIALS=[PATH]`. Singular requests only require the API key and service URL. Singular authentication data is stored in the base event configuration so it is automatically set when constructing an event. The API key is added as a field to the data object in the HTTP request to provide authentication.

#### **4.2.2 Creating an endpoint**

Clients need to be able to send requests for events that require data not available on the server. Not many events need to be sent this way, but it is still best to create a single endpoint to handle all client event requests. API endpoints are communication channels that clients can call to access server resources. MoW implements a REST API that uses HTTP requests to transport data. REST includes four HTTP verbs to distinguish request types (GET, POST, PUT and DELETE), but only GET and POST are used for analytics. Data included in POST requests are sent in JSON format for quick and easy data handling on the server. The API is built around Express, a popular web application framework for Node.js.

REST API requests call predefined URL paths or routes to execute wanted operations. As seen in Figure 6, routes consist of three parts: a URL path, a HTTP verb and a handler function. In order to ensure as much backwards compatibility as possible, the Express router is used to separate routes based on the client version number. Separating functionality based on client version also enables creating code in advance and merely activating it by changing the version number when needed. The endpoint in Figure 6 is a POST action that takes an object containing event data, applies required processing and

sends the results to BigQuery. URL for the endpoint is combined from a prefix from the router and the given parameter “/events”.

```
27 router.post('/events', authentication.auth, function(req, res, next) {
28   req.checkBody('bundle', 'Invalid bundle').notEmpty();
29   let errors = req.validationErrors();
30   if (errors) {
31     response.send(res, metadata.basic(GLOBAL.status.param.missing), response.errorMsg(errors));
32   } else {
33     let bundle = req.body.bundle;
34     constructInstance.processAndSend(bundle)
35       .then(function (result) {
36         console.log(result);
37         response.send(res, metadata.basic(GLOBAL.status.success), result);
38       })
39       .catch(function (error) {
40         response.send(res, metadata.basic(error.code), response.msg(error.message));
41       })
42   }
43 }
44 });
```

Figure 6 API route for events sent by a client

Before executing any operation, the sending user is authenticated by comparing the device ID, player ID and token headers against database values. If a valid token is not found for the user, the authentication fails, and a response is sent immediately (Figure 7). As soon as a valid token is found, next() is called to skip the rest of the function and continue to execute the main function of that route.

```

147 authentication.auth = function (req, res, next) {
148   var _token = req.get(GLOBAL.header.request.token);
149   var playerId;
150   if(req.get(GLOBAL.header.request.user)) {
151     playerId = req.get(GLOBAL.header.request.user);
152   }else{
153     playerId = req.params.id;
154   }
155   var deviceId = req.get(GLOBAL.header.request.dev);
156
157   if((typeof deviceId !== 'undefined') && (typeof _token !== 'undefined')) {
158     var verification = token.verify(deviceId, _token);
159     if((verification === token.VALID) || (verification === token.EXPIRING)) {
160       if(GLOBAL.auth_mode_db) {
161         TokenModel.findOne({playerId:playerId,token:_token}).lean().exec(function (error, docs) {
162           if(error) {
163             res.status(GLOBAL.status.server_error);
164             console.log('DB ERROR ' + error);
165             response.send(res, metadata.basic(GLOBAL.status.token.invalid), response.msg('database error'));
166           }else if(docs) {
167             next();
168           }else{
169             res.status(GLOBAL.status.token.invalid);
170             console.log('AUTH- token not found ');
171             response.send(res, metadata.basic(GLOBAL.status.token.invalid), response.msg('token-id not found'));
172           }
173         });
174       }else{
175         next();
176       }
177     }else{
178       res.status(GLOBAL.status.token.invalid);
179       response.send(res, metadata.basic(GLOBAL.status.token.invalid), response.msg('invalid token'));
180     }
181   }else{
182     console.log('## FAILS HERE ');
183     res.status(GLOBAL.status.token.required);
184     response.send(res, metadata.basic(GLOBAL.status.token.required), response.msg(GLOBAL.responseMsg.missingParam));
185   }
186 };

```

Figure 7 Player token authentication function

In addition to the general endpoint for client events, another endpoint was created for updating some mandatory player data. When a player launches the game, the client will call this endpoint and send some information about the used device, client version and platform. This information is passed to a handler function that locates the player's IP-address using the "geoip-lite" node module and saves the data as a database document (Figure 8). An entirely new document is created if one cannot be found with the supplied player ID. If one already exists, it is overwritten with new data.

```

269  /**
270   * handle: updates player analytics information when the game starts
271   * @param req
272   * @return {Promise<any>}
273   */
274  handle (req) {
275      return new Promise(function (resolve, reject) {
276          Construct.getGeolocation(req)
277              .then(function (result) {
278                  let bundle = req.body.analytics;
279                  bundle.ip = result.ip;
280                  bundle.playerCountry = result.country;
281                  return analyticsQuery.create(bundle);
282              })
283              .then(function (result) {
284                  return resolve(result);
285              })
286              .catch(function (error) {
287                  console.log('ERROR @Client.handle ' + error);
288                  return reject(error);
289              })
290      });
291  }

```

Figure 8 Mandatory analytics handler function

### 4.2.3 Processing event data

All data that is sent to BigQuery has to meet some structural requirements. Events that do not match the pre-set format are declined and will not be saved. For this implementation, all events follow the same format as mentioned when creating the BigQuery table. The main reason and advantage for using a single event schema inside BigQuery is that event structures can be better customised and modified on the fly. Events can be configured separately in backend code even though they all have the same base structure.

Event structures are configured under their own file seen in Figure 8. The configuration specifies which fields of the source data should be included in the event data. Because all events are configured in the same location, modifying their structure is effortless. In case someone wanted to exclude “cardRarity” from the “cardGained” event in Figure 9, all that needs to be done is to remove that line from the schema. Apart from common data that is included in every event, anything that is not specified here is not added to the event. The event configuration does not search for data that is missing from the source, it is only used to exclude all unnecessary fields. Whoever calls to send an event is responsible for providing enough source data for processing.

```
2 //Sent whenever a card is gained in any manner
3 let cardGained = {
4   cardLevel: Number,
5   cardName: String,
6   cardRank: Number,
7   cardRarity: String,
8   cardSource: String,
9   cardType: String,
10  cardAmount: Number,
11  currentOfficer: String,
12  currentOfficerLevel: Number
13 };
14
15 //Sent whenever a card is upgraded
16 let cardUpgraded = {
17   cardLevelFrom: Number,
18   cardLevelTo: Number,
19   cardName: String,
20   cardRank: Number,
21   cardRarity: String,
22   cardType: String,
23   currentOfficer: String,
24   currentOfficerLevel: Number
25 };
```

Figure 9 Event schema configuration

Event requests sent from the client go through a slightly different process than ones sent directly from the server. Separate handling is necessary due to the client-side implementation of the analytics system. It is made possible for the client to send event requests that contain multiple events in bulk. Because the request data might contain data for more than one event, the sent events must be first extracted from a bundle. Found event data is looped through and then constructed in the same way as all others. The “processAndSend” function takes the data sent by a client and prepares it for sending. The data bundle first has its structure validated to make sure it can be properly handled. Then the events included in the bundle are looped through recursively and each one is passed on to the “constructAndSend” function. All is done asynchronously since the client does not need to know if the events were sent or not. If any errors occur during the process, they are logged into the console. (Figure 10.)

```

25 |   processAndSend (bundle) {
26 |       let config = [];
27 |       let self = this;
28 |       let player;
29 |       log.info('Processing analytics with bundle', bundle);
30 |       return new Promise(function (resolve, reject) {
31 |           validator.validate([{schema: CONSTANT.schema.clientProcessParams, data: bundle}])
32 |               .then(function (result) {
33 |                   let events = bundle.events;
34 |                   function loopItems(q) {
35 |                       if (q >= events.length) {
36 |                           return resolve(config);
37 |                       }
38 |                       let event = events[q];
39 |                       self.constructAndSend(event, event.name, bundle.playerId)
40 |                       loopItems(q + 1);
41 |                   }
42 |                   loopItems(0);
43 |               })
44 |               .catch(function (error) {
45 |                   console.log('ERROR @Client.process: '+error);
46 |                   return reject(error);
47 |               });
48 |       });
49 |   }

```

Figure 10 processAndSend function

Analytics events that are sent directly from the server do not go through this separation and directly call “constructAndSend”. “ConstructAndSend” is a wrapper function used to construct events based on their configuration and send them to BigQuery. The caller of this function is responsible for providing appropriate data for construction. As seen in Figure 11, the function takes in three arguments: a data bundle, name of the event and player ID. PlayerId is added to the data bundle which is then given to the construction function along with the given event name.

```

170 |   constructAndSend(bundle, eventName, playerId){
171 |       let self = this;
172 |       bundle.playerId = playerId;
173 |       return new Promise(function (resolve, reject){
174 |           self.constructEvent(bundle, eventName)
175 |               .then(function(data){
176 |                   let tableName = Construct.getTable_name();
177 |                   if (tableName.toUpperCase() === 'LOCAL')
178 |                       {
179 |                           tableName = 'test';
180 |                       }
181 |
182 |                   return resolve(transportInstance.sendEvents(tableName, data));
183 |               })
184 |               .catch(function(error){
185 |                   log.error('ERROR@ constructAndSend' + error);
186 |                   return reject(error);
187 |               });
188 |       });
189 |   }

```

Figure 11 constructAndSend function



Once the event data has been constructed, it is passed over to the "send-Events" function with the target destination table name. For testing purposes, if the current server environment value is "LOCAL", the target destination table name is manually set. (Figure 11.) The target table name is configured via the server environment variable to keep testing, development and production data separate.

```

191     constructEvent(bundle, eventName){
192         let data = [];
193         let player;
194         let self = this;
195
196         return new Promise(function (resolve, reject){
197             Construct.getPlayers({_id: bundle.playerId})
198             .then(function (result){
199                 player = result[0];
200                 return Promise.all([
201                     Construct.getAnalytics({playerId: bundle.playerId}),
202                     Construct.getPlayerCurrency({_id: bundle.playerId}, {gold: 1, gems: 1}),
203                     Construct.addCurrentOfficerInfoToBundle(player, bundle)
204                 ]);
205             })
206             .then(function(promises){
207                 let analytics = promises[0];
208                 let currency = Construct.currencies(promises[1]);
209                 bundle = promises[2];
210                 let item = self.eventMandatoryData(player, analytics, currency, eventName);
211
212                 let config = Schemas[eventName];
213                 let keys = Object.keys(config);
214
215                 for (var i = 0; i < keys.length; i++)
216                 {
217                     if (bundle[keys[i]])
218                     {
219                         item.eventProperties[keys[i]] = bundle[keys[i]];
220                     }
221                 }
222
223                 item.eventProperties = JSON.stringify(item.eventProperties);
224                 data.push(item);
225                 return resolve(data);
226             })
227             .catch(function (error){
228                 log.error('ERROR @' + eventName + ': ' + error);
229                 return reject(error);
230             });
231         });
232     }

```

Figure 12 constructEvent function

Before sending an event, one must be constructed using the given data. Constructing an event can be divided into three sections: fetching common event data, adding mandatory data and adding event specific data. Figure 12 demonstrates how an event is constructed using the source data and the event name. Lines 197 through 204 contain the first section; fetching common event data. Any additional data that needs to be included in all events can be

added to the list of Promises to return here. After gathering all wanted data, the data is passed to the next function in which the promises are extracted into their own objects. A base event object is created from the gathered data using “eventMandatoryData”. This function searches the database for some important player specific data and uses it with the previously fetched data to create an event object. Finally, the event configuration is used to loop through the events fields. Bundle fields that have a matching name are added to the event object, everything else is ignored. Lastly, the event object is converted into a JSON string format ready to be sent to BigQuery and returned to the wrapper function. (Figure 12.)

```

31     sendEvents (tableName, rows) {
32         log.info('Sending following analytics to table ' + tableName, rows);
33         return new Promise(function (resolve, reject) {
34             bigquery
35                 .dataset(datasetId)
36                 .table(tableName)
37                 .insert(rows)
38                 .then((results) => {
39                     log.info(`Inserted ${rows.length} rows @ ` + tableName);
40                     return resolve(results)
41                 })
42                 .catch(err => {
43                     if (err) {
44                         if (err.name === 'PartialFailureError')
45                         {
46                             if (err.errors && err.errors.length > 0) {
47                                 err.errors.forEach(err => log.error(err));
48                             }
49                         }
50                         else {
51                             log.error('ERROR:', err);
52                         }
53                     }
54                     return reject(err)
55                 });
56         });
57     }
58 }

```

Figure 13 sendEvents function

Events that are returned and ready to be sent are passed to the “sendEvents” function seen in Figure 13. Event data is given as separate rows to be inserted to BigQuery with their target destination table name. The function utilises Google’s own library to insert rows into a table and returns errors found during the process. Some logging was added to provide a clear understanding of what is being inserted and where.

#### 4.2.4 Singular events

Singular is used to track purchases made by players inside the game and other events that could be useful in player profiling. Player profiling enables the use of targeted special offers with content that is interesting and relevant for the player. All Singular events are paired with a normal analytics event as purchase data has interest in analysis elsewhere as well.

Like normal analytics, Singular events are also sent from the server. A few events are sent directly from the client and unlike normal analytics, they do not pass through the backend server but use the Singular Unity SDK instead. Server-side events are sent via a REST API provided by Singular for server to server integration. Events use a specific GET endpoint at <https://s2s.singular.net/api/v1/evt?> to record any custom events needed (Singular 2018). Singular events have many required parameters that must be included in the parameters object sent to the endpoint. Some parameters are optional, but the events cannot be fully customised in terms of fields. Details about the required/optional parameters can be seen in Appendix 1.

In order to make sure Singular events have all required parameters assigned, they are built from a set base event configuration. Figure 14 describes the construction of a server-side singular event. Mandatory analytics data is fetched from the database as with normal analytics, but the data is assigned to required parameters. Additional event data can be provided to the function to be added as an optional field seen on line 35. After the simple construction, the event is sent to the Singular endpoint using axios, a node module for HTTP requests (Figure 15).

```

29     event (playerId, eventData, eventName) {
30         return new Promise(function (resolve, reject) {
31             let event = CONSTANT.event;
32             Construct.getAnalytics({playerId: playerId})
33                 .then(function (analytic) {
34                     event.n = eventName;
35                     event.e = JSON.stringify(eventData);
36                     event.p = analytic.platform;
37                     event.ip = analytic.ip;
38                     event.ve = analytic.osVersion;
39                     event.ma = analytic.deviceManufacturer;
40                     event.mo = analytic.deviceName;
41                     event.lc = analytic.localeTag;
42                     event.uptime = moment().utc().unix().toString();
43                     event.app_v = analytic.clientVersion;
44
45                     let platform = analytic.platform.toUpperCase();
46                     if (platform === 'IOS') {
47                         event.idfv = analytic.vendorId;
48                         event.idfa = analytic.advertisingID;
49                     } else if (platform === 'ANDROID') {
50                         event.aifa = analytic.advertisingID;
51                     }
52                     return Event.send(EVENT_URL, event);
53                 })
54                 .then(function (result) {
55                     return resolve(result);
56                 })
57                 .catch(function (error) {
58                     return reject(error);
59                 })
60             });
61     }

```

Figure 14 Singular event construction function

```

16     static send (URL, data) {
17         log.debug('Sending singular event to ' + URL + ' containing ' + JSON.stringify(data, null, 2));
18         return new Promise(function (resolve, reject) {
19             axios.get(URL, {params: data})
20                 .then(function (response) {
21                     return resolve(response);
22                 })
23                 .catch(function (error) {
24                     return reject(error);
25                 });
26         });
27     }
28 }

```

Figure 15 Singular event send function

Revenue events sent to Singular have some additional requirements to them. Revenue events use the same API endpoint as other events but must be

named correctly and include a few of the earlier optional parameters. Purchase events have their own construction method with slightly changed functionality. Additional event data is not added as a parameter in contrast to normal Singular events. The event must have the name “\_\_IAP\_\_”, an ISO 4217 currency code and the currency amount. This data is added under the fields “n”, “cur” and “amt” respectively. (Singular 2018.)

#### 4.2.5 Server event placement

Once the basic event construction and sending framework is implemented, the events themselves must be placed. As mentioned before, all events call the same construction function with different parameters. All that needs to be done is create a configuration for the event, gather all the data it requires and provide it to the constructing function. Implementing new events is very fast and straightforward due to building a generic framework.

```

889     async.each(cards, function (card, cb) {
890         PlayerCardsManager.addCardToPlayer(player._id, card.card, card.amount, function(error){
891             // Construct and send cardGained event
892             PlayerCardsManager.getPlayerCardData(player._id, card.card, function(error, bundle){
893                 if (error)
894                 {
895                     cb(error);
896                 }
897                 else
898                 {
899                     bundle.cardSource = chestName;
900                     bundle.cardAmount = card.amount;
901                     constructInstance.constructAndSend(bundle, 'cardGained', player._id)
902                         .catch(function(error){
903                             log.error('ANALYTICS ERROR @cardGained: ' + error);
904                         });
905                     cb();
906                 }
907             });
908         });
909     },
910     function (err) {
911         return callback(err);
912     });

```

Figure 16 Implementation of "cardGained" event

Figure 16 displays an implementation of the “cardGained” event that is called when a player receives cards as rewards from a chest. Analytics is added to the section where the player has received the card on lines 892 through 904. First, a helper function is called to fetch information about the given card and create a base bundle for the analytics function. A few extra fields are the added to the data and on line 901, the construction function is called. Any errors that would occur are caught and logged. Event implementations can be

much simpler than this example as well, only consisting of a couple lines of code. The complexity of the implementation completely depends on the amount and way to find the data required. Regardless of implementation, the core is always the same, calling the construction function with a data bundle, event name and player ID.

#### **4.2.6 Event testing**

After writing the implementation for all different events, they need to be tested. Testing must be systematic to ensure that all necessary data is coming through. During planning, an Excel sheet was created with all information related to event parameters, names and placement. Two checkboxes were also added to the sheet for every event to keep track of implementation and testing. The implemented checkbox would be checked once the event was presumably done and ready to be tested, the tested checkbox is then checked when confirmed.

As opposite to the original plan of reserving a separate time for testing the implemented events, events were tested upon implementation, one at a time. Testing a single event at a time whenever one is implemented provides flexibility and confirmation that no events are left incomplete. Events are tested by doing unit tests for the backend code and checking the correctness of data from the BigQuery console. After the unit tests have passed, the events are then finally tested in an actual environment by triggering the event in game. Figure 17 shows an example unit test for a server-side event. In the unit test shown below, the “chai” testing library is used to create test functions that are run in a loop. All test functions that are added to the utility array are executed when testing is run. Test functions are executed in order and as soon as the previous one returns, the next one is called.

```

33 function eventSchema(callback){
34     let eventName = 'cardGained';
35     let playerId = 'changeThisToPlayerId';
36     let bundle = {cardName:"testCard", cardRank:1, currentOfficerLevel:2, asd:2};
37     describe('send test event to BigQuery', function (){
38         it('should send the event', function(done){
39             constructInstance.constructAndSend(bundle, eventName, playerId)
40                 .then(function(success){
41                     done();
42                     callback();
43                 })
44                 .catch(function(err){
45                     done(err);
46                     callback();
47                 });
48         });
49     });
50 }
51
52 utility.add([
53     eventSchema
54 ]);
55 let Test = function () {};
56
57 // test runner
58 Test.run = function (callback) {
59     let suit = utility.get();
60     function executor(q) {
61         if (q >= suit.length) {
62             return callback('***** COMPLETE @analytics.test *****');
63         }
64
65         suit[q](function () {
66             executor(q + 1);
67         });
68     }
69
70     executor(0);
71 };

```

Figure 17 Server-side event unit test

In addition to server-side events, events sent from clients need to be tested. Figure 18 demonstrates a test function for an event that would be sent from the client. Again, the “chai” test library is used to mock a POST request with specific data and headers. For this test, a manual timeout must be set so that the test is not marked complete before the event is sent to BigQuery. Without a timeout, this test would be considered done as soon as the request response is processed. Due to the event sending implementation being asynchronous, clients not needing information in the response and the response being sent before the events are fully processed, a delay must be set manually. In contrast, the test function shown in Figure 17 does not implement a

manual wait because it waits for the “constructAndSend” function to return a Promise before being marked complete.

```

12 function send_events(callback) {
13   let bundle = {playerId:"changeThisToPlayerId",events:[{name: 'testEvent'}]}
14   describe('/POST analytics/events ', function () {
15     it('it should create player analytics bundle', function (done) {
16       Config.chai.request(Config.server)
17         .post(version + 'analytics/events')
18         .set('x-user', 'userHeader')
19         .set('x-dev', 'deviceHeader')
20         .set('x-token', 'tokenHeader')
21         .send({bundle: bundle })
22         .end(function (err, res) {
23           setTimeout(function (error) {
24             utility.print(true, err, res.body);
25             done();
26             callback();
27           }, 1000);
28         });
29     });
30   });
31 }

```

Figure 18 Client event request test function

### 4.3 Frontend implementation

This game analytics implementation focuses on handling everything in the backend, but that does not make the frontend implementation any less important. Client-side code processes mostly the sending of user and device data to the database and the triggering of some specific events. Events are processed and sent using an event queue that stores and sends events as a batch instead of sending events immediately. Storing events helps in having control over when event requests are sent to the server.

#### 4.3.1 Singular SDK

In order to send events relevant to marketing from the client, the Singular Unity SDK is used. Singular provides a software development kit for Unity developers as a Unity package. The Unity package can be imported to the project by downloading and opening the file from Singular’s website. To set up the SDK, an object is created into the main scene and the “SingularSDK” script is attached to it. In order to connect to the correct Singular project, an API Key and API Secret are set in the inspector. The SDK will initialise itself using the provided data when the game is started if the “Initialize On Awake”



is checked. Some additional steps are required for iOS and Android setup, which can be found in the SDK documentation. (Singular 2018.)

For this thesis, no custom client-side events are implemented for Singular as it is beyond the scope of this thesis. Singular is used to track user sessions and to register users for revenue tracking. Revenue tracking itself is done via backend code, therefore no additional actions are required on the client. For data privacy reasons, Singular offers a way to stop tracking a user and disable the SDK. Tracking can later be resumed with the user's consent. (Singular 2018.)

### 4.3.2 Event queue

Client-side events utilize an event queue to control sending event requests to the server. When an event is triggered, a method is called to record the event by adding event data into the queue (Figure 19). Event data is stored in JSON format to keep the queue processing simple. Sending events and removing them from the queue occurs on a set interval and sends up to a set maximum number of events at a time. Limiting the maximum number of sent events keeps the packet size small to avoid large network spikes.

```

98     /// <summary>
99     /// Adds the event name and adds the event data to queue
100    /// </summary>
101    /// <param name="eventName"></param>
102    /// <param name="eventData"></param>
103    private static void QueueData(string eventName, JsonObject eventData)
104    {
105        if (sendInterval <= 0.0f)
106        {
107            return;
108        }
109
110        eventData.AddField("name", eventName);
111
112        _eventQueue.Add(eventData);
113    }

```

Figure 19 Queue event method

Initially, the goal was to keep the event queue running as a coroutine, but that would require a `MonoBehaviour` object in the scene at all times. Having a dedicated object to process event sending is not desirable as the manager script requires an instance and running the coroutine on another object could cause

problems. So, instead of using a coroutine, a plugin called UniRx is used. UniRx offers a solution to running methods on an interval without linking them to an object. As seen in Figure 20, a “CompositeDisposable” and an “Observable” are used to solve the problem. An “Observable” provides a notification by calling a method on all observers that have subscribed to it when a certain condition is met (Microsoft 2017). A “CompositeDisposable” acts as an observer that subscribes to the provider and defining the method to call upon receiving a notification. In this case, whenever the interval condition in “Observable.IntervalFrame” is met, “EventSender()” is called and executed because “disposable” has subscribed to the observable with that method. Because the game runs at 30 frames per second, the interval is set to 30 times the set number of seconds.

```

11     internal static float sendInterval = 1.0f;
12     internal static int maxSendAmount = 10;
13
14     private static string _sessionId;
15     private static List<JsonObject> _eventQueue = new List<JsonObject>();
16
17     private static CompositeDisposable disposable = new CompositeDisposable();
18
19     /// <summary>
20     /// Adds the event sender as observable with an interval based subscription
21     /// </summary>
22     public static void StartSender()
23     {
24         Observable.IntervalFrame((int)(30 * sendInterval)).Subscribe(x => EventSender()).AddTo(disposable);
25     }
26
27     /// <summary>
28     /// Clears all subscriptions from the NGAalyticsSender disposable
29     /// </summary>
30     public static void ClearSender()
31     {
32         disposable.Clear();
33     }
34

```

Figure 20 Event queue sender subscription

Before events can be sent, the data must be prepared for the request. All events that are sent to the server as a request must contain the player ID, event name and the event data. Every event that is going to be sent on an interval is packed into a single JsonObject and passed on to the sending method. (Figure 21.)

```

117     /// <summary>
118     /// Creates a single bundle of queued events with a set maximum amount
119     /// </summary>
120     /// <returns></returns>
121     private static JSONObject PrepareData(bool sendAll = false)
122     {
123         if (_eventQueue.Count <= 0)
124         {
125             return null;
126         }
127
128         JSONObject data = new JSONObject();
129         data.AddField("playerId", ProfileManager.m_PlayerId);
130
131         JSONObject events = new JSONObject(JSONObject.Type.ARRAY);
132         data.AddField("events", events);
133
134         int queuedItems = 0;
135         foreach (JSONObject eventData in _eventQueue)
136         {
137             events.Add(eventData);
138             queuedItems++;
139             if (queuedItems > maxSendAmount && !sendAll)
140                 break;
141         }
142
143         JSONObject bundle = new JSONObject();
144         bundle.AddField("bundle", data);
145
146         return bundle;
147     }

```

Figure 21 PrepareData method

Once the data has been prepared for sending, it is passed on to the "Send-Events" method that removes the sent events from the queue and calls "NetworkRequestManager" to send the HTTP request. The response callback is set to be empty since the client does not need to know what happens after the event is sent. (Figure 22.)

```

65     /// <summary>
66     /// Wrapper for sending the event queue to the server
67     /// </summary>
68     private static void EventSender()
69     {
70         UnityEngine.Debug.Log("Event batch sent here. Event count: " + _eventQueue.Count);
71         SendEvents(PrepareData());
72     }
73
74     /// <summary>
75     /// Sends the given events to the server
76     /// </summary>
77     /// <param name="eventsData">Events to send</param>
78     private static void SendEvents(JSONObject eventsData, bool sendAll = false)
79     {
80         if (_eventQueue.Count <= 0 || eventsData == null)
81         {
82             return;
83         }
84
85         if (_eventQueue.Count > maxSendAmount && !sendAll)
86         {
87             _eventQueue.RemoveRange(0, maxSendAmount);
88         }
89         else
90         {
91             _eventQueue.Clear();
92         }
93
94         NetworkRequestManager.Instance.Request("/analytics/events", eventsData, ProfileManager.GetWWWHeader(), reply =>
95         {
96
97         });
98     }

```

Figure 22 Event sender methods

## 5 CONCLUSIONS

The purpose of this thesis was to describe the implementation process of a highly customisable backend analytics framework. It provides an insight to the implementation steps and processes that must be considered when developing an analytics system using BigQuery. The resulting product can be deemed a success as a modified version of this thesis is in development for another project.

The documentation of this game analytics implementation can be used as a guide on how to develop a server-side event-based analytics solution in conjunction with BigQuery. Even though analytics solutions are always project specific and this project is no different, the documentation was presented in as general terms as possible. Creating and setting up a BigQuery project, event sending pipeline, client event requests and event testing are separated into their own sections. The steps required to implement a customisable analytics framework are explained in order of implementation to make it easy to follow.

At the start of this thesis, with no prior experience or knowledge of backend code, Node.js or BigQuery, productivity was quite slow. Working effectiveness kept increasing gradually as new aspects were learned and applied into use.

Invaluable experience from both game analytics and server-side code was gained in the making of the analytics system that will surely prove useful in the future. During this thesis, many conversations were had that proved extremely useful and teaching. Senior programmers at Nitro Games provided constant support in learning and developing the system according to requirements.

Creating a general framework for processing and sending analytics events was important as it could then be used in future projects. The system can be copied over and reconfigured to fit another environment with ease, thus saving time for core game development. Much has been learned during development that can be used to further develop the analytics system in terms of scalability and optimisation.

## REFERENCES

Cooladata, 2017. *Building an Event-Based Data Model for Analyzing Online Data*. [Online]

Available at: <https://www.cooladata.com/building-event-based-data-model-analyzing-online-data/>

[Accessed 11 10 2018].

Datamatics, n.d. *Fear and Proactive Thinking: The main driver for Event-based Analytics*. [Online]

Available at: <https://www.datamatics.com/articles/fear-and-proactive-thinking-main-driver-event-based-analytics>

[Accessed 16 10 2018].

Egan, W., 2016. *What Is Event Based Marketing*. [Online]

Available at: <https://www.willegan.com/what-is-event-based-marketing/>

[Accessed 16 10 2018].

Fine, R., 2017. *UnityScript's long ride off into the sunset*. [Online]

Available at: <https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/>

[Accessed 13 11 2018].

Google, 2018. *Google BigQuery*. [Online]

Available at: <https://cloud.google.com/bigquery/>

[Accessed 16 10 2018].

Google, 2018. *Quotas & Limits*. [Online]

Available at: [https://cloud.google.com/bigquery/quotas#dataset\\_limits](https://cloud.google.com/bigquery/quotas#dataset_limits)

[Accessed 18 10 2018].

Google, 2018. *Understanding Service Accounts*. [Online]

Available at: <https://cloud.google.com/iam/docs/understanding-service-accounts>

[Accessed 19 10 2018].

McCulloch, L., 2016. *The Importance of Data Validation*. [Online]

Available at: <https://deltadna.com/blog/importance-data-validation/>

[Accessed 13 11 2018].

Microsoft, 2017. *Observer Design Pattern*. [Online]

Available at: <https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern>

[Accessed 21 11 2018].

Nitro Games Plc, 2018. *Medals of War*. [Online]

Available at: <https://www.medalsofwargame.com/>

[Accessed 3 10 2018].

Nitro Games Plc, 2018. *Nitro Games Investors*. [Online]  
Available at: <https://www.nitrogames.com/investors/>  
[Accessed 13 11 2018].

Orsini, L., 2013. *What You Need To Know About Node.js*. [Online]  
Available at: <https://readwrite.com/2013/11/07/what-you-need-to-know-about-nodejs/>  
[Accessed 13 11 2018].

RoboMQ, n.d. *Event Driven Analytics*. [Online]  
Available at: <https://www.robomq.io/products/eventdrivenanalytics.html>  
[Accessed 16 10 2018].

Singular, 2018. *Accurate Attribution For Mobile App Activities*. [Online]  
Available at: <https://www.singular.net/advanced-mobile-attribution/#integrations>  
[Accessed 13 11 2018].

Singular, 2018. *Server to Server Integration*. [Online]  
Available at: <https://developers.singular.net/docs/server-to-server-integration>  
[Accessed 01 11 2018].

Singular, 2018. *Singular Unity SDK*. [Online]  
Available at: <https://developers.singular.net/docs/unity-sdk>  
[Accessed 21 11 2018].

**FIGURES**

- Figure 1 Creating the BigQuery project
- Figure 2 BigQuery project view
- Figure 3 Creating a BigQuery dataset
- Figure 4 Creating a BigQuery data table
- Figure 5 Downloading a service account key
- Figure 6 API route for events sent by a client
- Figure 7 Player token authentication function
- Figure 8 Mandatory analytics handler function
- Figure 9 Event schema configuration
- Figure 10 processAndSend function
- Figure 11 constructAndSend function
- Figure 12 constructEvent function
- Figure 13 sendEvents function
- Figure 14 Singular event construction function
- Figure 15 Singular event send function
- Figure 16 Implementation of "cardGained" event
- Figure 17 Server-side event unit test
- Figure 18 Client event request test function
- Figure 19 Queue event method
- Figure 20 Event queue sender subscription
- Figure 21 PrepareData method
- Figure 22 Event sender methods



## ATTACHMENTS

Attachment 1/1

## Required Event Parameters

Parameter	Description	Supported Platforms	Example
n	Name of the event.	iOS,Android	ViewItem
a	Singular API Key.	iOS,Android	a42be1d8119389dd36c7acbeaf6a
p	Platform Android or iOS.	iOS,Android	Android
i	Longname (Android) or Bundle ID (iOS) of your application.	iOS,Android	com.singular.app
ip	The IP of the session.	iOS,Android	10.1.2.3
ve	OS Version of the device at session time.	iOS,Android	9.2
ma	Make of the device hardware, typically the consumer-facing name (e.g. Samsung, LG, Apple). This parameter must be used with the model parameter.	iOS,Android	samsung
mo	Model of the device hardware (e.g. iPhone 4S, Galaxy SIII). This parameter must be used with the make parameter.	iOS,Android	SM-G935F
lc	The IETF local tag for the device, using two-letter language and country code separated by an underscore.	iOS,Android	en_US
bd	Build of the device, URL encoded	iOS,Android	Build%2F13D15
idfa	For iOS apps only. Upper-case raw <u>advertising ID</u> with dashes.	iOS	DFC5A647-9043-4699-B2A5-76F03A970648
idfv	For iOS apps only. Upper-case raw <u>IdentifierForVendor</u> with dashes.	iOS	21D86612-0983-4ECC-84AC-B35380AF1334
aifa	For Android apps only. Lower-case raw <u>advertising ID</u> with dashes.	Android	8ecd7512-2864-440c-93f3-a3cabe62525b
andi	For Android apps only. Lower-case raw <u>android ID</u> . Required only when Android Advertising ID is not available on the device.	Android	fc8d449516de0dfb

## Optional Event Parameters

Parameter	Description	Supported Platforms	Example
utime	Time of the session in UNIX time	iOS,Android	1483228800
amt	For revenue events. The currency amount expressed as a numerical value with decimal points (e.g. 2.51). This should be used in conjunction with the cur parameter.	iOS,Android	2.51
cur	For revenue events. The <a href="#">ISO 4217</a> three-letter currency code. This should be used in conjunction with the amt parameter.	iOS,Android	EUR
umilisec	Time of the session in milliseconds UNIX time.	iOS,Android	1483228800000
e	Event attributes formatted as a JSON encoded value.	iOS,Android	%7B%22item_name%22%3A%20%22Box%22
use_ip	Extract the IP field from the HTTP request instead of the 'ip' field. Don't use this with the ip parameter	iOS,Android	false