GCM Architectural Overview

Google Cloud Messaging for Android (GCM) is a free service that helps developers send data from servers to their Android applications on Android devices, and upstream messages from the user's device back to the cloud. This could be a lightweight message telling the Android application that there is new data to be fetched from the server (for instance, a movie uploaded by a friend), or it could be a message containing up to 4kb of payload data (so apps like instant messaging can consume the message directly). The GCM service handles all aspects of queueing of messages and delivery to the target Android application running on the target device.

GCM introduces GCM Cloud Connection Server (CCS), which you can use in tandem with GCM HTTP service/endpoint/APIs. CCS uses XMPP, and it offers asynchronous, bidirectional messaging. For more information, see GCM Cloud Connection Server (ccs.html).

> To jump right into using GCM with your Android applications, see the instructions in Getting Started (gs.html).

QUICKVIEW

- Get an introduction to key GCM terms and concepts.
- Learn the basic features of a GCM application.
- Understand the role of the 3rd-party application server, and how to send messages and process results.

IN THIS DOCUMENT

Introduction

Here are the primary characteristics of Google Cloud Messaging (GCM):

- It allows 3rd-party application servers to send messages to their Android applications.
- Using the GCM Cloud Connection Server, you can receive upstream messages from the user's device.
- An Android application on an Android device doesn't need to be running to receive messages. The system will wake up the Android application via Intent broadcast when the message arrives, as long as the application is set up with the proper broadcast receiver and permissions.
- It does not provide any built-in user interface or other handling for message data. GCM simply passes raw message data received straight to the Android application, which has full control of how to handle it. For example, the application might post a notification, display a custom user interface, or silently sync data.
- It requires devices running Android 2.2 or higher that also have the Google Play Store application installed, or or an emulator running Android 2.2 with Google APIs. However, you are not limited to deploying your Android applications through Google Play Store.
- It uses an existing connection for Google services. For pre-3.0 devices, this requires users to set up their Google account on their mobile devices. A Google account is not a requirement on devices running Android 4.0.4 or higher.

Architectural Overview

This section gives an overview of how GCM works.

This table summarizes the key terms and concepts involved in GCM. It is divided into these categories:

- **Components** — The physical entities that play a role in GCM.
- **Credentials** — The IDs and tokens that are used in different stages of GCM to ensure that all parties have been authenticated, and that the message is going to the correct place.

Components	
Mobile Device	The device that is running an Android application that uses GCM. This must be a 2.2 Android device that has Google Play Store installed, and it must have at least one logged in Google

	account if the device is running a version lower than Android 4.0.4. Alternatively, for testing you can use an emulator running Android 2.2 with Google APIs.
3rd-party Application Server	An application server that developers set up as part of implementing GCM in their applications. The 3rd-party application server sends data to an Android application on the device via the GCM server.
GCM Servers	The Google servers involved in taking messages from the 3rd-party application server and sending them to the device.
	Credentials
Sender ID	A project number you acquire from the API console, as described in Getting Started. The sender ID is used in the registration process to identify an Android application that is permitted to send messages to the device.
Application ID	The Android application that is registering to receive messages. The Android application is identified by the package name from the manifest. This ensures that the messages are targeted to the correct Android application.
Registration ID	An ID issued by the GCM servers to the Android application that allows it to receive messages. Once the Android application has the registration ID, it sends it to the 3rd-party application server, which uses it to identify each device that has registered to receive messages for a given Android application. In other words, a registration ID is tied to a particular Android application running on a particular device. **Note:** If you use backup and restore, you should explicitly avoid backing up registration IDs. When you back up a device, apps back up shared prefs indiscriminately. If you don't explicitly exclude the GCM registration ID, it could get reused on a new device, which would cause delivery errors.
Google User Account	For GCM to work, the mobile device must include at least one Google account if the device is running a version lower than Android 4.0.4.
Sender Auth Token	An API key that is saved on the 3rd-party application server that gives the application server authorized access to Google services. The API key is included in the header of POST requests that send messages.
Notification Key	Part of the user notifications feature, which provides a mapping between a user and instances of an app running on multiple devices owned by the user. The `notification_key` is the token that GCM uses to fan out notifications to all devices whose registration IDs are associated with the key. For more discussion of this topic, see User Notifications.
Notification Key Name	Part of the user notifications feature. The `notification_key_name` is a name or identifier (can be a username for a 3rd-party app) that is unique to a given user. It is used by third parties to group together registration IDs for a single user. For more discussion of this topic, see User Notifications.

Lifecycle Flow

- Enable GCM. An Android application running on a mobile device registers to receive messages.
- User Notifications. A 3rd-party server can optionally group multiple registration IDs in a `notification_key` to send messages to multiple devices owned by a single user.
- Send a message. A 3rd-party application server sends messages to the device.
- Receive a message. An Android application receives a message from a GCM server.

These processes are described in more detail below.

Enable GCM

This is the sequence of events that occurs when an Android application running on a mobile device registers to receive messages:

1. The first time the Android application needs to use the messaging service, it fires off a registration Intent to a GCM server.

   This registration Intent (`com.google.android.c2dm.intent.REGISTER`) includes the sender ID, and the Android application ID.

> **Note:** Because there is no lifecycle method that is called when the application is run for the first time, the registration intent should be sent on `onCreate()`, but only if the application is not registered yet.

2. If the registration is successful, the GCM server broadcasts a `com.google.android.c2dm.intent.REGISTRATION` intent which gives the Android application a registration ID.

   The Android application should store this ID for later use (for instance, to check on `onCreate()` if it is already registered). Note that Google may periodically refresh the registration ID, so you should design your Android application with the understanding that the `com.google.android.c2dm.intent.REGISTRATION` intent may be called multiple times. Your Android application needs to be able to respond accordingly.

3. To complete the registration, the Android application sends the registration ID to the application server. The application server typically stores the registration ID in a database.

The registration ID lasts until the Android application explicitly unregisters itself, or until Google refreshes the registration ID for your Android application.

> **Note:** When users uninstall an application, it is not automatically unregistered on GCM. It is only unregistered when the GCM server tries to send a message to the device and the device answers that the application is uninstalled or it does not have a broadcast receiver configured to receive `com.google.android.c2dm.intent.RECEIVE` intents. At that point, your server should mark the device as unregistered (the server will receive a `NotRegistered (#unreg_device)` error).

Note that it might take a few minutes for the registration ID to be completely removed from the GCM server. So if the 3rd-party server sends a message during this time, it will get a valid message ID, even though the message will not be delivered to the device.

Send a Message

For an application server to send a message to an Android application, the following things must be in place:

- The Android application has stored a target that it can specify as the recipient of a message. This can be one of the following:
  - A single registration ID (or an array of registration IDs) that allows the app to receive messages for a particular device.
  - A `notification_key` and corresponding `notification_key_name`, used to map a single user to multiple registration IDs. For more discussion of this topic, see User Notifications.
- An API key. This is something that the developer must have already set up on the application server for the Android application (for more discussion, see Role of the 3rd-party Application Server). Now it will get used to send messages to the device.

Here is the sequence of events that occurs when the application server sends a message:

1. The application server sends a message to GCM servers.
2. Google enqueues and stores the message in case the device is offline.
3. When the device is online, Google sends the message to the device.
4. On the device, the system broadcasts the message to the specified Android application via Intent broadcast with proper permissions, so that only the targeted Android application gets the message. This wakes the Android application up. The Android application does not need to be running beforehand to receive the message.
5. The Android application processes the message. If the Android application is doing non-trivial processing, you may want to grab a `PowerManager.WakeLock` and do any processing in a Service.

An Android application can unregister GCM if it no longer wants to receive messages.

Receive a Message

This is the sequence of events that occurs when an Android application installed on a mobile device receives a message:

1. The system receives the incoming message and extracts the raw key/value pairs from the message payload, if any.

2. The system passes the key/value pairs to the targeted Android application in a `com.google.android.c2dm.intent.RECEIVE` Intent as a set of extras.
3. The Android application extracts the raw data from the `com.google.android.c2dm.intent.RECEIVE` Intent by key and processes the data.

What Does the User See?

When mobile device users install Android applications that include GCM, the Google Play Store will inform them that the Android application includes GCM. They must approve the use of this feature to install the Android application.

Role of the 3rd-party Application Server

Before you can write client Android applications that use the GCM feature, you must have an application server that meets the following criteria:

- Able to communicate with your client.
- Able to fire off HTTPS requests to the GCM server.
- Able to handle requests and resend them as needed, using exponential back-off.
- Able to store the API key and client registration IDs. The API key is included in the header of POST requests that send messages.

Sending Messages

This section describes how the 3rd-party application server sends messages to one or more mobile devices. Note the following:

- A 3rd-party application server can either send messages to a single device or to multiple devices. A message sent to multiple devices simultaneously is called a *multicast message*.
- To send a single message to multiple devices owned by a single user, you can use a `notification_key`, as described in User Notifications.
- You have 2 choices in how you construct requests and responses: plain text or JSON.
- However, to send multicast messages, you must use JSON. Plain text will not work.

Before the 3rd-party application server can send a message to an Android application, it must have received a registration ID from it.

Request format

To send a message, the application server issues a POST request to `https://android.googleapis.com/gcm/send`.

A message request is made of 2 parts: HTTP header and HTTP body.

The HTTP header must contain the following headers:

- `Authorization`: key=YOUR_API_KEY
- `Content-Type`: `application/json` for JSON; `application/x-www-form-urlencoded;charset=UTF-8` for plain text.

For example:

```
Content-Type:application/json
Authorization:key=AIzaSyB-1uEai2WiUapxCs2Q0GZYzPu7Udno5aA

{
  "registration_ids" : ["APA91bHun4MxP5egoKMwt2KZFBaFUH-1RYqx..."],
  "data" : {
    ...
  },
}
```

Note: If `Content-Type` is omitted, the format is assumed to be plain text.

The HTTP body content depends on whether you're using JSON or plain text. For JSON, it must contain a string representing a JSON object with the following fields:

Field	Description
`registration_ids`	A string array with the list of devices (registration IDs) receiving the message. It must contain at least 1 and at most 1000 registration IDs. To send a multicast message, you must use JSON. For sending a single message to a single device, you could use a JSON object with just 1 registration id, or plain text (see below). A request must include a recipient—this can be either a registration ID, an array of registration IDs, or a `notification_key`.
`notification_key`	A string that maps a single user to multiple registration IDs associated with that user. This allows a 3rd-party server to send a single message to multiple app instances (typically on multiple devices) owned by a single user. A 3rd-party server can use `notification_key` as the target for a message instead of an individual registration ID (or array of registration IDs). The maximum number of members allowed for a `notification_key` is 10. For more discussion of this topic, see User Notifications. Optional.
`notification_key_name`	A name or identifier (can be a username for a 3rd-party app) that is unique to a given user. It is used by 3rd parties to group together registration IDs for a single user. The `notification_key_name` should be uniquely named per app in case you have multiple apps for the same project ID. This ensures that notifications only go to the intended target app. For more discussion of this topic, see User Notifications.
`collapse_key`	An arbitrary string (such as "Updates Available") that is used to collapse a group of like messages when the device is offline, so that only the last message gets sent to the client. This is intended to avoid sending too many messages to the phone when it comes back online. Note that since there is no guarantee of the order in which messages get sent, the "last" message may not actually be the last message sent by the application server. See Advanced Topics for more discussion of this topic. Optional.
`data`	A JSON object whose fields represents the key-value pairs of the message's payload data. If present, the payload data it will be included in the Intent as application data, with the key being the extra's name. For instance, `"data":{"score":"3x1"}` would result in an intent extra named `score` whose value is the string `3x1`. There is no limit on the number of key/value pairs, though there is a limit on the total size of the message (4kb). The values could be any JSON object, but we recommend using strings, since the values will be converted to strings in the GCM server anyway. If you want to include objects or other non-string data types (such as integers or booleans), you have to do the conversion to string yourself. Also note that the key cannot be a reserved word (`from` or any word starting with `google.`). To complicate things slightly, there are some reserved words (such as `collapse_key`) that are technically allowed in payload data. However, if the request also contains the word, the value in the request will overwrite the value in the payload data. Hence using words that are defined as field names in this table is not recommended, even in cases where they are technically allowed. Optional.
`delay_while_idle`	If included, indicates that the message should not be sent immediately if the device is idle. The server will wait for the device to become active, and then only the last message for each `collapse_key` value will be sent. Optional. The default value is `false`, and must be a JSON boolean.
`time_to_live`	How long (in seconds) the message should be kept on GCM storage if the device is offline. Optional (default time-to-live is 4 weeks, and must be set as a JSON number).
`restricted_package_name`	A string containing the package name of your application. When set, messages will only be sent to registration IDs that match the package name. Optional.
`dry_run`	If included, allows developers to test their request without actually sending a

	message. Optional. The default value is `false`, and must be a JSON boolean.

If you are using plain text instead of JSON, the message fields must be set as HTTP parameters sent in the body, and their syntax is slightly different, as described below:

Field	Description
`registration_id`	Must contain the registration ID of the single device receiving the message. Required.
`collapse_key`	Same as JSON (see previous table). Optional.
`data.<key>`	Payload data, expressed as parameters prefixed with `data.` and suffixed as the key. For instance, a parameter of `data.score=3x1` would result in an intent extra named `score` whose value is the string `3x1`. There is no limit on the number of key/value parameters, though there is a limit on the total size of the message. Also note that the key cannot be a reserved word (`from` or any word starting with `google.`). To complicate things slightly, there are some reserved words (such as `collapse_key`) that are technically allowed in payload data. However, if the request also contains the word, the value in the request will overwrite the value in the payload data. Hence using words that are defined as field names in this table is not recommended, even in cases where they are technically allowed. Optional.
`delay_while_idle`	Should be represented as `1` or `true` for `true`, anything else for `false`. Optional. The default value is `false`.
`time_to_live`	Same as JSON (see previous table). Optional.
`restricted_package_name`	Same as JSON (see previous table). Optional.
`dry_run`	Same as JSON (see previous table). Optional.

If you want to test your request (either JSON or plain text) without delivering the message to the devices, you can set an optional HTTP or JSON parameter called `dry_run` with the value `true`. The result will be almost identical to running the request without this parameter, except that the message will not be delivered to the devices. Consequently, the response will contain fake IDs for the message and multicast fields (see Response format (#response)).

Example requests

Here is the smallest possible request (a message without any parameters and just one recipient) using JSON:

```
{ "registration_ids": [ "42" ] }
```

And here the same example using plain text:

```
registration_id=42
```

Here is a message with a payload and 6 recipients:

```
{ "data": {
    "score": "5x1",
    "time": "15:10"
  },
  "registration_ids": ["4", "8", "15", "16", "23", "42"]
}
```

Here is a message with all optional fields set and 6 recipients:

```
{ "collapse_key": "score_update",
  "time_to_live": 108,
  "delay_while_idle": true,
```

```
  "data": {
    "score": "4x8",
    "time": "15:16.2342"
  },
  "registration_ids":["4", "8", "15", "16", "23", "42"]
}
```

And here is the same message using plain-text format (but just 1 recipient):

```
collapse_key=score_update&time_to_live=108&delay_while_idle=1&data.score=4x8&data.time
```

> **Note:** If your organization has a firewall that restricts the traffic to or from the Internet, you need to configure it to allow connectivity with GCM in order for your Android devices to receive messages. The ports to open are: 5228, 5229, and 5230. GCM typically only uses 5228, but it sometimes uses 5229 and 5230. GCM doesn't provide specific IPs, so you should allow your firewall to accept outgoing connections to all IP addresses contained in the IP blocks listed in Google's ASN of 15169.

Response format

There are two possible outcomes when trying to send a message:

- The message is processed successfully.
- The GCM server rejects the request.

When the message is processed successfully, the HTTP response has a 200 status and the body contains more information about the status of the message (including possible errors). When the request is rejected, the HTTP response contains a non-200 status code (such as 400, 401, or 503).

The following table summarizes the statuses that the HTTP response header might contain. Click the troubleshoot link for advice on how to deal with each type of error.

Response	Description
200	Message was processed successfully. The response body will contain more details about the message status, but its format will depend whether the request was JSON or plain text. See Interpreting a success response for more details.
400	Only applies for JSON requests. Indicates that the request could not be parsed as JSON, or it contained invalid fields (for instance, passing a string where a number was expected). The exact failure reason is described in the response and the problem should be addressed before the request can be retried.
401	There was an error authenticating the sender account. Troubleshoot
5xx	Errors in the 500-599 range (such as 500 or 503) indicate that there was an internal error in the GCM server while trying to process the request, or that the server is temporarily unavailable (for example, because of timeouts). Sender must retry later, honoring any `Retry-After` header included in the response. Application servers must implement exponential back-off. Troubleshoot

Interpreting a success response

When a JSON request is successful (HTTP status code 200), the response body contains a JSON object with the following fields:

Field	Description
`multicast_id`	Unique ID (number) identifying the multicast message.
`success`	Number of messages that were processed without an error.
`failure`	Number of messages that could not be processed.
`canonical_ids`	Number of results that contain a canonical registration ID. See Advanced Topics for more discussion of this topic.
	Array of objects representing the status of the messages processed. The objects are listed in

`results`
: the same order as the request (i.e., for each registration ID in the request, its result is listed in the same index in the response) and they can have these fields:
- `message_id`: String representing the message when it was successfully processed.
- `registration_id`: If set, means that GCM processed the message but it has another canonical registration ID for that device, so sender should replace the IDs on future requests (otherwise they might be rejected). This field is never set if there is an error in the request.
- `error`: String describing an error that occurred while processing the message for that recipient. The possible values are the same as documented in the above table, plus "Unavailable" (meaning GCM servers were busy and could not process the message for that particular recipient, so it could be retried).

If the value of `failure` and `canonical_ids` is 0, it's not necessary to parse the remainder of the response. Otherwise, we recommend that you iterate through the results field and do the following for each object in that list:

- If `message_id` is set, check for `registration_id`:
  - If `registration_id` is set, replace the original ID with the new value (canonical ID) in your server database. Note that the original ID is not part of the result, so you need to obtain it from the list of `registration_ids` passed in the request (using the same index).
- Otherwise, get the value of `error`:
  - If it is `Unavailable`, you could retry to send it in another request.
  - If it is `NotRegistered`, you should remove the registration ID from your server database because the application was uninstalled from the device or it does not have a broadcast receiver configured to receive `com.google.android.c2dm.intent.RECEIVE` intents.
  - Otherwise, there is something wrong in the registration ID passed in the request; it is probably a non-recoverable error that will also require removing the registration from the server database. See Interpreting an error response for all possible error values.

When a plain-text request is successful (HTTP status code 200), the response body contains 1 or 2 lines in the form of key/value pairs. The first line is always available and its content is either `id=`*`ID of sent message`* or `Error=`*`GCM error code`*. The second line, if available, has the format of `registration_id=`*`canonical ID`*. The second line is optional, and it can only be sent if the first line is not an error. We recommend handling the plain-text response in a similar way as handling the JSON response:

- If first line starts with `id`, check second line:
  - If second line starts with `registration_id`, gets its value and replace the registration IDs in your server database.
- Otherwise, get the value of `Error`:
  - If it is `NotRegistered`, remove the registration ID from your server database.
  - Otherwise, there is probably a non-recoverable error (**Note:** Plain-text requests will never return `Unavailable` as the error code, they would have returned a 500 HTTP status instead).

Interpreting an error response

Here are the recommendations for handling the different types of error that might occur when trying to send a message to a device:

Missing Registration ID
: Check that the request contains a registration ID (either in the `registration_id` parameter in a plain text message, or in the `registration_ids` field in JSON).
Happens when error code is `MissingRegistration`.

Invalid Registration ID
: Check the formatting of the registration ID that you pass to the server. Make sure it matches the registration ID the phone receives in the `com.google.android.c2dm.intent.REGISTRATION` intent and that you're not truncating it or adding additional characters.
Happens when error code is `InvalidRegistration`.

Mismatched Sender
: A registration ID is tied to a certain group of senders. When an application registers for GCM usage, it must specify which senders are allowed to send messages. Make sure you're using one of those when trying to send messages to the device. If you switch to a different sender, the existing registration IDs

won't work. Happens when error code is `MismatchSenderId`.

Unregistered Device

An existing registration ID may cease to be valid in a number of scenarios, including:

- If the application manually unregisters by issuing a `com.google.android.c2dm.intent.UNREGISTER` intent.
- If the application is automatically unregistered, which can happen (but is not guaranteed) if the user uninstalls the application.
- If the registration ID expires. Google might decide to refresh registration IDs.
- If the application is updated but the new version does not have a broadcast receiver configured to receive `com.google.android.c2dm.intent.RECEIVE` intents.

For all these cases, you should remove this registration ID from the 3rd-party server and stop using it to send messages.

Happens when error code is `NotRegistered`.

Message Too Big

The total size of the payload data that is included in a message can't exceed 4096 bytes. Note that this includes both the size of the keys as well as the values.

Happens when error code is `MessageTooBig`.

Invalid Data Key

The payload data contains a key (such as `from` or any value prefixed by `google.`) that is used internally by GCM in the `com.google.android.c2dm.intent.RECEIVE` Intent and cannot be used. Note that some words (such as `collapse_key`) are also used by GCM but are allowed in the payload, in which case the payload value will be overridden by the GCM value.

Happens when the error code is `InvalidDataKey`.

Invalid Time To Live

The value for the Time to Live field must be an integer representing a duration in seconds between 0 and 2,419,200 (4 weeks). Happens when error code is `InvalidTtl`.

Authentication Error

The sender account that you're trying to use to send a message couldn't be authenticated. Possible causes are:

- Authorization header missing or with invalid syntax.
- Invalid project number sent as key.
- Key valid but with GCM service disabled.
- Request originated from a server not whitelisted in the Server Key IPs.

Check that the token you're sending inside the `Authorization` header is the correct API key associated with your project. You can check the validity of your API key by running the following command:

```
# api_key=YOUR_API_KEY

# curl --header "Authorization: key=$api_key" --header Content-Type:"application/
```

If you receive a 401 HTTP status code, your API key is not valid. Otherwise you should see something like this:

```
{"multicast_id":6782339717028231855,"success":0,"failure":1,"canonical_ids":0,"re
```

If you want to confirm the validity of a registration ID, you can do so by replacing "ABC" with the registration ID.

Happens when the HTTP status code is 401.

Timeout

The server couldn't process the request in time. You should retry the same request, but you MUST obey the following requirements:

- Honor the `Retry-After` header if it's included in the response from the GCM server.
- Implement exponential back-off in your retry mechanism. This means an exponentially increasing delay after each failed retry (e.g. if you waited one second before the first retry, wait at least two second before the next one, then 4 seconds and so on). If you're sending multiple messages, delay each one independently by an additional random amount to avoid issuing a new request for all messages at the same time.

Senders that cause problems risk being blacklisted.

Happens when the HTTP status code is between 501 and 599, or when the `error` field of a JSON object in the results array is `Unavailable`.

Internal Server Error
: The server encountered an error while trying to process the request. You could retry the same request (obeying the requirements listed in the Timeout section), but if the error persists, please report the problem in the android-gcm group.
Happens when the HTTP status code is 500, or when the `error` field of a JSON object in the results array is `InternalServerError`.

Invalid Package Name
: A message was addressed to a registration ID whose package name did not match the value passed in the request. Happens when error code is `InvalidPackageName`.

Example responses

This section shows a few examples of responses indicating messages that were processed successfully. See Example requests (#example-requests) for the requests these responses are based on.

Here is a simple case of a JSON message successfully sent to one recipient without canonical IDs in the response:

```
{ "multicast_id": 108,
  "success": 1,
  "failure": 0,
  "canonical_ids": 0,
  "results": [
    { "message_id": "1:08" }
  ]
}
```

Or if the request was in plain-text format:

```
id=1:08
```

Here are JSON results for 6 recipients (IDs 4, 8, 15, 16, 23, and 42 respectively) with 3 messages successfully processed, 1 canonical registration ID returned, and 3 errors:

```
{ "multicast_id": 216,
  "success": 3,
  "failure": 3,
  "canonical_ids": 1,
  "results": [
    { "message_id": "1:0408" },
    { "error": "Unavailable" },
    { "error": "InvalidRegistration" },
    { "message_id": "1:1516" },
    { "message_id": "1:2342", "registration_id": "32" },
    { "error": "NotRegistered"}
  ]
}
```

In this example:

- First message: success, not required.
- Second message: should be resent (to registration ID 8).
- Third message: had an unrecoverable error (maybe the value got corrupted in the database).
- Fourth message: success, nothing required.
- Fifth message: success, but the registration ID should be updated in the server database (from 23 to 32).
- Sixth message: registration ID (42) should be removed from the server database because the application was uninstalled from the device.

Or if just the 4th message above was sent using plain-text format:

```
Error=InvalidRegistration
```

If the 5th message above was also sent using plain-text format:

```
id=1:2342
registration_id=32
```

Viewing Statistics

To view statistics and any error messages for your GCM applications:

1. Go to the `Developer Console`.
2. Login with your developer account.

   You will see a page that has a list of all of your apps.

3. Click on the "statistics" link next to the app for which you want to view GCM stats.

   Now you are on the statistics page.

4. Go to the drop-down menu and select the GCM metric you want to view.

> **Note:** Stats on the Google API Console are not enabled for GCM. You must use the Developer Console (http://play.google.com/apps/publish).