Nakama Unity Client

Nakama is an open-source distributed social and realtime server for games and apps.

It includes a large set of services for users, data storage, and realtime client/server communication; as well as specialized APIs like realtime multiplayer, groups/guilds, and chat.

The Unity client handles all realtime client/server communication with Nakama. It supports all Nakama server features which include users, data storage, groups, realtime multiplayer, realtime chat, and more.

Install Server

Start by downloading Nakama, and then follow these pages to get started:

1. Install Nakama
2. Configure your setup
3. Start the server

The rest of this guide talks about setting up the Nakama Unity Client to communicate with the Nakama server.

Getting starting with Unity Client

In the Unity editor create a new C# script. This can be done via the Assets menu with "Assets > Create > C# Script". Modify the initial code in the script to connect with the server:

TIP	**Run with a local server** These instructions assume you've got a Nakama server setup and started with default connection details "127.0.0.1" and port "7350".

```
using UnityEngine;
using System.Collections;
using Nakama;

public class NewBehaviourScript : MonoBehaviour {
  void Start() {
    INClient client = NClient.Default("defaultkey");
  }

  void Update() {
  }
}
```

An `INClient` object is created which represents a connection to the server. Have a read of Controlling GameObjects Using Components for examples on how to share a C# object across your game objects.

You can change all the default connection settings with an `NClient.Builder`:

```
INClient client = new NClient.Builder("defaultkey")
    .Host("myprivate.server.com")
    .Port(8080)
    .SSL(true)
    .Build();
```

Authenticate

The `INClient` manages a socket connection with the server but before you can connect to the socket you must authenticate.

The device ID option is the most simple register/login option for users. It is done entirely without user interaction, so the user experience is completely frictionless. You can use `SystemInfo.deviceUniqueIdentifier` which is guaranteed to be unique on every device.

```
client.Register(request, (INSession session) => {
  // cache session on device
}, (INError error) => {
  Debug.LogErrorFormat("ID register '{0}' failed: {1}", uid, error);
});
```

If the user has already registered with the device ID you should use:

```
var uid = SystemInfo.deviceUniqueIdentifier;
var request = NAuthenticateMessage.Device(uid);
client.Login(request, (INSession session) => {
  // cache session on device
}, (INError error) => {
  Debug.LogErrorFormat("ID login '{0}' failed: {1}", uid, error);
});
```

TIP **Session cache** It's a good idea to cache a session on a device and restore it at game startup to avoid asking the user to login or register again.

Both "login" and "register" give us an `INSession` object. This session is used to establish a connection with the server.

```
client.Connect(session, (bool connected) => {
  Debug.Log("Socket connected.");
});
```

TIP	**Session lifecycle** Use `client.Disconnect()` when the game client loses focus or is closed and reconnect when it's active. See the Awake and Start tutorial on the Unity website.

User

The core component of every client interaction with Nakama is the user. A user represents a unique identity in Nakama, and can have one or more unique identifiers attached - device IDs, email addresses, Facebook accounts, and more.

Authenticate

In the Unity client, an `INClient` can be constructed with the server key like below:

```
private static readonly string ServerKey = "YourServerKey";
private readonly INClient client;

public MyClass() {
  client = new NClient.Builder(ServerKey).Build();
}
```

Once you are authenticated, you'll need to connect the `client` instance to the server.

```
// First obtain a session:
// client.Register(...) or client.Login(...)
// We recommend caching the resulting `session` locally.
// Then connect:
client.Connect(session);
```

There are three main login mechanisms so far; ID-based Login, Social Login, and Email Login. Currently supported Social Login providers include Facebook, Google, Steam and GameCenter. You have the option of providing any combination of login options.

Device ID

The most effortless way of authenticating a user with Nakama is to use a Device ID. This can be done entirely without user interaction, so the user experience is completely frictionless.

TIP	**Device ID login as guest users.** They don't require any information from the user to create although you'll want to use an identifier from the client device which can be used to recover the user if necessary.

Anonymous logins are performed by sending Nakama a single field representing a unique identifier, usually a UDID or a Device ID. This unique identifier can be used again later to retrieve the same user for example when recovering the user after the game is removed and reinstalled.

TIP	**Cache Device ID** The UDID from mobile devices like Android and iOS are changed often by the operating system vendors. They are not reliable sources for uniquely identifying the device; you should cache the UDID once obtained so the game can use it to recover the account.

In the Unity Client, device ID authentication is done as below once you have an instance of `INClient`:

```
// Let's check storage if there was an ID already generated
// if not, let's generate a new one and persist it.
// Alternatively, on mobile devices you may use a native system device ID.
string id = PlayerPrefs.GetString("ID");
if (string.IsNullOrEmpty (id)) {
  id = SystemInfo.deviceUniqueIdentifier;
  PlayerPrefs.SetString ("ID", id);
}

var message = NAuthenticateMessage.Device(id);
client.Register(message, (INSession newSession) =>
{
  client.Connect(session);
  Debug.Log ("Successfully logged in.");
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
});
```

If the user has already registered with the device ID you should use:

```
var request = NAuthenticateMessage.Device(id);
client.Login(request, (INSession session) => {
  client.Connect(session);
  Debug.Log ("Successfully logged in.");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
});
```

Email

Email based login is for users that still want to log in, but would prefer not to connect through their social or platform accounts. Users are registered and logged in via an email address and password.

Registration

```
string email = "email@example.com"
string password = "password"
var message = NAuthenticateMessage.Email(email, password);
client.Register(message, (INSession newSession) =>
{
  client.Connect(session);
  Debug.Log ("Successfully logged in.");
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not register user: '{0}'.", error.Message);
});
```

Login

```
string email = "email@example.com"
string password = "password"
var message = NAuthenticateMessage.Email(email, password);
client.Login(message, (INSession newSession) =>
{
  client.Connect(session);
  Debug.Log ("Successfully logged in.");
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
});
```

Social Login

Nakama provides a simple mechanism for your game to implement social login, allowing your gamers to sign in using Facebook, Google, Steam or GameCenter and start playing right away.

To perform a social login, ensure the gamer is first logged into their social account. Once the social authentication is complete, the last step is to send Nakama the access token available. This is usually an OAuth 2.0 access token, but varies by social provider.

Facebook

You will need the Facebook Unity SDK which can be downloaded here. Follow the Facebook Unity Examples on how to add the asset to your project. You must also complete the instructions on Facebook's developer guide on how to configure your iOS or Android client.

You'd need to ensure that the Facebook profile has been registered with Nakama before attempting to login with it.

```
// you must call FB.Init as early as possible at game startup
if (!FB.IsInitialized) {
```

```
  FB.Init (() => {
    if (FB.IsInitialized) {
      FB.ActivateApp();
      // Use a Facebook access token to create a user account
      var oauthToken = Facebook.Unity.AccessToken.CurrentAccessToken.TokenString;
      var message = NAuthenticateMessage.Facebook(oauthToken);
      client.Login(message, (INSession session) =>
      {
        client.Connect(session);
        Debug.Log ("Successfully logged in.");
      }, (INError error) =>
      {
        Debug.Log ("Could not login. Attempting to register.");
        client.Register(message, (INSession session) =>
        {
          client.Connect(session);
          Debug.Log ("Successfully registered and logged in.");
        }, (INError error) =>
        {
          Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
        });
      });
    }
  });
}

// Execute in a button or UI component within your game
FB.Login("email", (ILoginResult result) => {
  if (FB.IsLoggedIn) {
    var accessToken = Facebook.Unity.AccessToken.CurrentAccessToken.TokenString;
    client.Login(message, (INSession session) =>
    {
      client.Connect(session);
      Debug.Log ("Successfully logged in.");
    }, (INError error) =>
    {
      Debug.Log ("Could not login. Attempting to register.");
      client.Register(message, (INSession session) =>
      {
        client.Connect(session);
        Debug.Log ("Successfully registered and logged in.");
      }, (INError error) =>
      {
        Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
      });
    });
  } else {
    Debug.LogErrorFormat ("Could not login to Facebook got '{0}'.", result.Error);
  }
});
```

Google

Similar to Facebook, authenticating via Google requires you to have an OAuth `AccessToken`. Once you have obtained the `AccessToken` you can use it to register and login to Nakama.

```
String oauthToken = "access-token-from-google";
var message = NAuthenticateMessage.Google(oauthToken);
client.Login(message, (INSession session) =>
{
  client.Connect(session);
  Debug.Log ("Successfully logged in.");
}, (INError error) =>
{
  Debug.Log ("Could not login. Attempting to register.");
  client.Register(message, (INSession session) =>
  {
    client.Connect(session);
    Debug.Log ("Successfully registered and logged in.");
  }, (INError error) =>
  {
    Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
  });
});
```

Steam

To authenticate with Steam, you first need a Steam `SessionToken` for the user. Once you have obtained the `SessionToken` you can use it to register and login to Nakama.

```
string sessionToken = "session-token-from-steam";
var message = NAuthenticateMessage.Steam(sessionToken);
client.Login(message, (INSession session) =>
{
  client.Connect(session);
  Debug.Log ("Successfully logged in.");
}, (INError error) =>
{
  Debug.Log ("Could not login. Attempting to register.");
  client.Register(message, (INSession session) =>
  {
    client.Connect(session);
    Debug.Log ("Successfully registered and logged in.");
  }, (INError error) =>
  {
    Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
  });
});
```

Game Center

Nakama supports authentication using Game Center Player IDs on compatible Apple devices. This is a good frictionless authentication option as it requires no user input.

Users can be authenticated by sending the following Game Center credentials to the server: Player ID, Bundle ID, Timestamp, Salt, Signature, and Public Key URL. You'll need to dive into native Objective-C code as the `UnityEngine.SocialPlatforms.GameCenter` doesn't expose enough information to enable authentication.

TIP	**Required parameters** Have a look at the relevant iOS GameKit function reference.

```
// These are passed in via your native Objective-C code...
string playerId;
string bundleId;
long timestamp;
string base64salt;
string base64signature;
string publicKeyUrl;

var message = NAuthenticateMessage.GameCenter(playerId, bundleId, timestamp,
base64salt, base64signature, publicKeyUrl);
client.Login(message, (INSession session) =>
{
  client.Connect(session);
  Debug.Log ("Successfully logged in.");
}, (INError error) =>
{
  Debug.Log ("Could not login. Attempting to register.");
  client.Register(message, (INSession session) =>
  {
    client.Connect(session);
    Debug.Log ("Successfully registered and logged in.");
  }, (INError error) =>
  {
    Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
  });
});
```

Custom

You can use a custom ID to authenticate users with Nakama. This is particularly useful if you have an external user identity service and would like to mirror the user IDs used in your system into Nakama.

Similar to Social Login, ensure that your custom ID has been registered with Nakama before attempting to login with it.

```
string customId = "your-custom-id";
var message = NAuthenticateMessage.Custom(customId);
client.Login(message, (INSession session) =>
{
  client.Connect(session);
  Debug.Log ("Successfully logged in.");
}, (INError error) =>
{
  Debug.Log ("Could not login. Attempting to register.");
  client.Register(message, (INSession session) =>
  {
    client.Connect(session);
    Debug.Log ("Successfully registered and logged in.");
  }, (INError error) =>
  {
    Debug.LogErrorFormat ("Could not login user: '{0}'.", error.Message);
  });
});
```

Link / Unlink

Linking allows the user to login using more than one type of identifier. It is very similar to the registration process for each authentication type. You can only link credentials that are not already in use by another user.

The user needs to be logged in and have a connected session with the server.

```
string id = "id";
var message = SelfLinkMessage.Device(id);
client.Send(message, (bool completed) => {
  Debug.Log ("Successfully linked device ID to current user");
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not link device ID: '{0}'.", error.Message);
});
```

To unlink, simply tell Nakama to remove the credentials:

```
string id = "id";
var message = SelfUnlinkMessage.Device(id);
client.Send(message, (bool completed) => {
  Debug.Log ("Successfully unlinked device ID from current user");
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not unlink device ID: '{0}'.", error.Message);
});
```

Fetch Self

The client can retrieve the currently logged-in user data from Nakama. This data includes common fields such as handle, fullname, avatar URL and timezone. Self will also include user's login information such as a list of device IDs associated and their social IDs.

```
var message = NSelfFetchMessage.Default();
client.Send(message, (INSelf self) => {
  Debug.LogFormat ("The user's ID is '{0}'.", self.Id);
  Debug.LogFormat ("The user's fullname is '{0}'.", self.Fullname); // may be null
  Debug.LogFormat ("The user's handle is '{0}'.", user.Handle);
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not retrieve self: '{0}'.", error.Message);
});
```

Update Self

The client can update the information stored about the currently logged-in user, such as their handle, fullname, location, or lang.

```
var message = new NSelfUpdateMessage.Builder()
                    .AvatarUrl("http://graph.facebook.com/avatar_url")
                    .Fullname("My New Name")
                    .Lang("en")
                    .Location("San Francisco")
                    .Timezone("Pacific Time")
                    .Build();
client.Send(message, (bool completed) => {
  Debug.Log ("Successfully updated user information");
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not update self: '{0}'.", error.Message);
});
```

Fetch Users

Nakama can give the client common information about other users. The client needs to know the IDs of those users.

TIP	**Public user information** Use this to display public user profiles, identify opponents in matches, and more.

```
var message = NUsersFetchMessage.Default(id);
client.Send(message, (INResultSet<INUser> results) => {
  Debug.LogFormat ("Fetched {0} users'.", results.Results.Count);
  foreach (INUser user in results.Results) {
    Debug.LogFormat ("The user's handle is '{0}'.", user.Handle);
  }
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not retrieve users: '{0}'.", error.Message);
});
```

Storage

Storage is a distributed key-value store. It can be used to store individual user data, maintain global data sets or configuration values, share data and user generated content, and much more.

A key is used to look up data from Storage and it is the composed of a bucket, collection and a record. Records are grouped into collections which are grouped further into buckets. You can create any number of records, collections and buckets.

Objects are JSON data stored against a key and must be less than 8KB in size. An object is identified by a key and (optionally) an owner.

Buckets are used to group or namespace data. Each bucket contains data identified by pair of collections and records, and each bucket enforces uniqueness of key-owner pairs. This means there will only ever be one object with a particular key, owned by a particular user, in any given bucket.

Permissions

Records can optionally assign individual permissions to decide how both the owner and others can interact with this data. Objects can only be modifiable by their owner. These permissions can be combined freely. The server will enforce read and write permissions independently.

Read permissions can be:

- `0` - The object cannot be read by either the owner or any other users.
- `1` - The object can only be read by the owner.
- `2` - The object is readable by any user.

Write permissions can be:

- `0` - The object is read-only.
- `1` - The object can be written or updated by its owner.

Write

Clients can only write to objects which belong to the user with the current session. A write request to a key that does not exist will implicitly create it so you don't need to check if the object exists before a write operation. Objects must be valid JSON.

Storage objects can be written and rewritten at any time, as often as the Client requires. If the key already exists this operation will correctly preserve original creation timestamps.

```
string bucket = "testBucket";
string collection = "testCollection";
string record = "testRecord";
byte[] storageValue = Encoding.UTF8.GetBytes("{\"jsonkey\":\"jsonvalue\"}");

var builder = new NStorageWriteMessage.Builder();
builder.Write(bucket, collection, record, storageValue);
// builder.Write(bucket, collection, record2, storageValue2); -- You can batch write
messages.

var message = builder.Build();
client.Send(message, (bool completed) => {
  Debug.Log ("Successfully storage data.");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not store data into storage: '{0}'.", error.Message);
});
```

Batch Write

Client can optionally batch write multiple objects to multiple keys in Nakama. Nakama provides transactional guarantees over a given batch write of data - If the input parameters are not expected or database insertion fails, the entire operation aborts.

```
var builder = new NStorageWriteMessage.Builder();
builder.Write(bucket, collection, record, storageValue);
builder.Write(bucket, collection, record2, storageValue2);
builder.Write(bucket, collection, record3, storageValue3);
var message = builder.Build();
```

Conditional Write

All stored objects are versioned as they are stored to the database. The server can accept simple version lookup queries that are very similar to HTTP ETag conditional headers.

This allows the Client to send the latest version information that it has locally alongside the data that needs to be updated in one request. The server will then validate to see if the versions are matched and if so will update the stored data.

There are two types of conditional writes:

1. If-Match: Client data version and server data version must match before the data is updated. This allows the Client to safely assume that it has the latest version of the data. The value is sent by the server and cached locally from a previous data fetch operation.
2. If-None-Match: This ensures that the Client does not overwrite data that is already stored on the server. This is useful for storage operations that only need to be done once. The only acceptable value is a "*".

```
var builder = new NStorageWriteMessage.Builder();
byte[] version; // This is an object version cached locally on the client.

// This is an If-Match check.
builder.Write(bucket, collection, record, storageValue, version);

// This is an If-None-Match check.
builder.Write(bucket, collection, record, storageValue, Encoding.UTF8.GetBytes("*"));

var message = builder.Build();
```

Fetch

Clients performing a read request can retrieve Storage objects identified by key and owner. An object with a null owner is referred to as global data.

If the object permissions allow it, a complete Storage object will be returned to the client.

```
string bucket = "testBucket";
string collection = "testCollection";
string record = "testRecord";
byte[] userId; // this value can be retrieve by sending a Self message.

var message = new NStorageFetchMessage.Builder().Fetch(bucket, collection, record,
userId).Build();
client.Send(message, (INResultSet<INStorageData> results) =>
  foreach (INStorageData data in results) {
    Debug.LogFormat ("Storage Bucket: '{0}', Collection: '{1}', Record: '{2}'", data
.Bucket, data.Collection, data.Record);
  }
}, (INError error) => {
  Debug.LogErrorFormat ("Could not fetch data from storage: '{0}'.", error.Message);
});
```

Remove

Objects can be deleted by their owners at any time with valid write permissions. Any request to delete keys that do not exist will succeed by default.

You can also conditionally remove an object if the object version matches the version available on the client.

```
string bucket = "testBucket";
string collection = "testCollection";
string record = "testRecord";
byte[] version; // This is an object version cached locally on the client.

var builder = new NStorageRemoveMessage.Builder();
builder.Remove(bucket, collection, record, version);

client.Send(message, (bool completed) => {
  Debug.Log ("Successfully removed data.");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not delete data from storage: '{0}'.", error.Message);
});
```

A delete operation performs a soft-delete on the server - data is not purged from the server but is no longer available to the client.

Friends

Friends are a core social feature in Nakama. Users can track new and existing friends, see who is online or when they were last seen, chat in real time, challenge them to matches, and more.

Each individual user's friends list is their primary way to track relationships to other individual users. These relationships can be:

- Mutual friend - both users have agreed to add each other as friends.
- Received invite - another user has sent a friend request.
- Sent invitation - an outgoing friend request from this user.
- Blocked - a blacklisted user. Used to reject chat requests, and more.

TIP	**Social accounts and friends** When registering or linking a social account, such as Facebook, Nakama will import friends automatically.

Add

Any user can add another user as a friend, identified by their user ID. If there is a pending friend invitation from that user, it will be accepted and the relationship between the two users will become mutual friendship.

Sending a friend invite and accepting a received invitation are done through the same client call.

```
// `id` is the user ID to add as a friend, or send an invite to.
client.Send<bool>(NFriendAddMessage.Default(id), (bool done) =>
{
  // Handle success.
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not add friend: '{0}'.", error.Message);
});
```

List

Listing friends for the current user is a single request which will return all their friends, regardless of relationship type.

When examining the returned list of friends, use `.State()` to differentiate the relationship types. State will be one of:

- `FriendState.Friend` - a mutual friend.
- `FriendState.Invite` - a received invite.
- `FriendState.Invited` - a sent invitation.

- `FriendState.Blocked` - a blocked user.

```
client.Send<bool>(NFriendsListMessage.Default(), (INResultSet<INFriend> results) =>
{
  foreach (INFriend friend in results) {
    Debug.LogFormat ("Friend ID '{0}'", friend.Id);
  }
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not list friends: '{0}'.", error.Message);
});
```

Remove

At any time users can decide to remove a mutual friend, reject a received invite, cancel a sent invitation, or unblock a user. All of these operations are done through the same client call.

TIP	**Re-adding removed friends** If a friend is removed and then re-added, they will have to accept the invitation again.

```
// `id` is the user ID to remove from friends list, or unblock.
client.Send<bool>(NFriendRemoveMessage.Default(id), (bool done) =>
{
  // Handle success.
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not remove friend: '{0}'.", error.Message);
});
```

Block

Nakama will observe the list of blocked users when other users trigger interactions. For example incoming direct chat messages or friend requests from blocked users will not be allowed.

```
// `id` is the user ID to block.
client.Send<bool>(NFriendBlockMessage.Default(id), (bool done) =>
{
  // Handle success.
}, (INError error) =>
{
  Debug.LogErrorFormat ("Could not block: '{0}'.", error.Message);
});
```

Groups

Groups connect a set of users and give them a shared space to interact through online presence tracking, persistent chat, and more. Users can join groups with their friends, new people they meet, or by looking for groups in their area.

Membership to a group grants privileges like access to a private group chat topic, chat history, and more.

Groups have **members** and **admins**. Both participate in group activities equally but admins have additional privileges and responsibilities. All users joining a group do so as regular members and may be promoted later by existing admins.

Join

Any user can choose to join a group, which will be processed by Nakama one of two ways:

- With **public** groups, the user is immediately added as a group member without further confirmation.
- For **private** groups, a join request is created and submitted to group admins for review. The user is not added as a member unless that request is accepted by any of the group's current admins.

```
byte[] groupId = null; // -- Get this by fetching a list of groups. See below.

var message = NGroupJoinMessage.Default(groupId);
client.Send(message, (bool completed) =>
{
  Debug.Log ("Successfully joined the group"); // -- If group is private, an join-
request invitation is sent
}, (INError error) => {
  Debug.LogErrorFormat ("Could not join group: '{0}'.", error.Message);
});
```

Users can list groups they've already joined. This provides fast access to all memberships for each user for further operations or individual group access.

Finding groups

Group listings allow users to find new groups to join based on activity, location, and more.

Nakama provides 3 main ways to find groups:

- By creation time - allowing users to find established groups first.
- By member count and updated at - to find recently active groups up to a given size.

- By lang tag and member count - to find groups in the user's area/country/timezone or primarily speaking the same language.

With these core building blocks users can quickly find one or more relevant groups to join and interact with based on interests, how frequently members are available, preferred language, and more.

```
var message = new NGroupListsMessage.Builder()
  .OrderByAsc(true)
  .FilterByLang("en")
  .Build();

client.Send(message, (INResultSet<INGroup> results) =>
{
  foreach (INGroup nakamaGroup in results) {
    Debug.LogFormat ("Group ID: '{0}'", nakamaGroup.Id);
  }
}, (INError error) => {
  Debug.LogErrorFormat ("Could not list groups: '{0}'.", error.Message);
});
```

TIP	**Group listing pagination** A cursor is returned with group listing results; use this in further calls to paginate results.

List joined groups

The Client can list the groups the user is part of. This list includes the groups the user is a member, an admin or has sent a request to join (for private groups).

```
var message = NGroupsSelfListMessage.Default();
client.Send(message, (INResultSet<INGroup> results) =>
{
  foreach (INGroup nakamaGroup in results) {
    Debug.LogFormat ("Group ID: '{0}'", nakamaGroup.Id);
  }
}, (INError error) => {
  Debug.LogErrorFormat ("Could not list groups: '{0}'.", error.Message);
});
```

Leave

Users can leave groups at their discretion. When the leave operation is accepted, the user immediately loses access to all group privileges.

```
byte[] groupId = null; // -- Get this by fetching a list of groups. See above.

var message = NGroupLeaveMessage.Default(groupId);
client.Send(message, (bool completed) =>
{
  Debug.Log ("Successfully left the group");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not leave group: '{0}'.", error.Message);
});
```

Create

To create a group users must choose a name and may submit other optional fields:

- Group Description
- Avatar URL
- Lang Tag (such as `en_US`)
- Additional application-specific metadata.

Nakama will use this data to create the group and ensure it appears in the correct listings based on its properties.

The user creating the group will automatically be added as a member with admin privileges.

```
var message = new NGroupCreateMessage
  .Builder("Group Name")
  .Description("Group Description")
  .Lang("en_US")
  .Private(true)
  .Build();

client.Send(message, (INGroup group) =>
{
  Debug.Log ("Successfully created a private group");
  Debug.LogFormat ("Group ID: {0}, Group Name: {1}", group.Id, group.Name);
}, (INError error) => {
  Debug.LogErrorFormat ("Could not create group: '{0}'.", error.Message);
});
```

Update

After a group is created, its admins may update it at any time. All group properties available at creation time can be edited later.

```
byte[] groupId = null; // -- Get this by fetching a list of groups. See above.

var message = new NGroupUpdateMessage
  .Builder(groupId)
  .Name("Updated Name")
  .Description("Updated Group Description")
  .Build();

client.Send(message, (bool completed) =>
{
  Debug.Log ("Successfully update group");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not update group: '{0}'.", error.Message);
});
```

Remove

Group admins are allowed to disband the group itself and remove all its members.

```
byte[] groupId = null; // -- Get this by fetching a list of groups. See above.

var message = NGroupRemoveMessage.Default(groupId);
client.Send(message, (bool completed) =>
{
  Debug.Log ("Successfully removed group");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not remove group: '{0}'.", error.Message);
});
```

Admins

Admins are group members with additional privileges, and are responsible for:

- Changing group config, description, and more.
- Approving or rejecting requests from other users to join the group.
- Removing members and admins.
- Appointing additional admins.

The user that creates a group is automatically assigned as the first admin, and may then promote others to admin status. All admins have equal privileges, and may remove other admins from the group.

TIP	**Leaving a group** Admins are allowed to leave groups just like regular members, with one exception: the last admin in a group cannot leave. They should promote at least one new admin from the other members, or remove and disband the group entirely.

Accepting join requests

When users attempt to join private groups a join request will be created. Admins can obtain a list of these requests and accept or reject each one. When a request is accepted that user becomes a member.

```
byte[] groupId = null; // -- Get this by fetching a list of groups. See above.
byte[] userId = null; // -- UserID of the member you'll like to accept or add to the
group.

var message = NGroupAddUserMessage.Default(groupId, userId);
client.Send(message, (bool completed) =>
{
  Debug.Log ("Successfully added user to group");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not add user to group: '{0}'.", error.Message);
});
```

Promote

Any admin may promote a regular group member to admin status. This ensures there is likely to be an admin presence available at most times to handle group moderation and leadership.

```
byte[] groupId = null; // -- Get this by fetching a list of groups. See above.
byte[] userId = null; // -- UserID of the member you'll like to accept or add to the
group.

var message = NGroupPromoteUserMessage.Default(groupId, userId);
client.Send(message, (bool completed) =>
{
  Debug.Log ("Successfully promoted user to admin");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not promote user to admin: '{0}'.", error.Message);
});
```

Kick

Admins can kick members or other admins from groups for any reason. This permanently removes that user from the group but does not prevent them from joining at a later point if needed.

```
byte[] groupId = null; // -- Get this by fetching a list of groups. See above.
byte[] userId = null; // -- UserID of the member you'll like to accept or add to the
group.

var message = NGroupKickUserMessage.Default(groupId, userId);
client.Send(message, (bool completed) =>
{
  Debug.Log ("Successfully kicked user from group");
}, (INError error) => {
  Debug.LogErrorFormat ("Could not kick user from group: '{0}'.", error.Message);
});
```

Realtime Chat

Nakama's **realtime chat** feature allows users to send messages directly to other individual users, to groups they belong to, or to open "named" chat rooms. These messages are delivered immediately over the socket to clients if the recipients are currently online, and stored in message history so offline users can catch up when they connect.

Topics

Chat topics are Nakama's way to identify message recipients. Topics tie together users that are currently online, serve as targets when sending messages, and tie together the history of messages sent through that topic.

Users explicitly join and leave topics each time they connect to Nakama. This allows users to selectively listen for messages on various topics, or opt for quiet periods where no messages are delivered while they're busy.

Users can join multiple topics at once from each connection to chat simultaneously in multiple groups or chat rooms.

TIP	**Multiple sessions** Topics only allow one concurrent connection from each user session, but the same user connected on multiple devices simultaneously can still join the same topic from each device.

Direct messages

Direct message (DM) topics represent 1-to-1 chat between two users. Any user can start a DM chat with other users in Nakama, their live messages and chat history are automatically kept private.

TIP	**Blocking users** See how to block users in Friends > Block to stop unwanted direct messages.

Group chat

Groups have individual private chat topics which only group members are allowed to join. Messages sent to these topics are propagated to other group members currently connected to the topic, and offline group members can replay message history when they next connect.

TIP	**Leaving groups** When a user leaves a group or is kicked, they lose access to the group's realtime chat topic and all message history.

Rooms

Chat rooms are dynamic spaces for any user to join and exchange messages. Rooms are identified by a name and are dynamically created when a user requests to join a room that did not previously

exist.

Presence

Nakama's **presence** system is the core mechanism used to track the users currently online on a given topic. Presences are identified by the user-session pair they belong to and are scoped to topics. This means a user connecting to the same topic from two different devices will have two visible presences on that topic, one for each session.

Presences can be used by clients to display a list of online group members, chat room participants, and more.

When a user successfully joins a topic, they receive the current presence list immediately. The server then sends presence updates periodically which contain any presences that have joined or left. No updates are sent if there are no changes to the presence list.

Sending messages

Users can send messages using the returned topic ID as a target once the topic is successfully joined. All message data is expected to be JSON encoded strings.

Senders receive an acknowledgement notification from the server for each successfully sent message containing the assigned message ID and other metadata.

The server will tag each message with a generated message ID, timestamp, and information about the user that sent it. All recipients can reliably use these metadata fields to identify messages as there is no way for users to set these fields on the client.

Receiving messages

Nakama will propagate messages to all users present on the topic when the message is sent. Messages are delivered in the order they are processed by the server.

Incoming message types

Alongside user chat messages, Nakama may insert additional messages into the chat stream. These are informational messages about users joining or leaving a group, being promoted to group admin, and more.

Message type	Source	Topics	Description
`0` (chat message)	User	All	Chat messages sent by users
`1` (group join)	Server	Group	Notification - a user joined the group
`2` (group add)	Server	Group	Notification - a user was added/accepted to the group

Message type	Source	Topics	Description
3 (group leave)	Server	Group	Notification - a user left the group
4 (group kick)	Server	Group	Notification - a user was kicked from the group
5 (group promoted)	Server	Group	Notification - a user was promoted to group admin

Message history

All chat topics automatically maintain a history of messages sent through them including notification messages generated by the server. Users with access to the topic can retrieve this history as needed to catch up on messages they missed while not connected.

Users can fetch message history starting from the most recent message and going backwards in time, or starting from the oldest available message and going forwards to current time. The server returns historic messages in batches alongside pagination cursors, which clients use to identify and retrieve the next set of messages in further fetch operations.

TIP **Message history without presence** Users can retrieve a topic's message history without joining effectively "peeking" at messages without subscribing for real-time delivery or appearing as a presence on the topic.

Licenses

Google.Protobuf.dll

This license applies to all parts of Protocol Buffers except the following:

- Atomicops support for generic gcc, located in src/google/protobuf/stubs/atomicops_internals_generic_gcc.h. This file is copyrighted by Red Hat Inc.
- Atomicops support for AIX/POWER, located in src/google/protobuf/stubs/atomicops_internals_power.h. This file is copyrighted by Bloomberg Finance LP.

Copyright 2014, Google Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Code generated by the Protocol Buffer compiler is owned by the owner of the input file used when generating it. This code is not standalone and requires a support library to be linked with it. This support library is itself covered by the above license.

WebSocket-Sharp License

The MIT License (MIT)

Copyright (c) 2010-2015 sta.blockhead

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Support

Join us on Gitter. This is the most immediate way to connect with Nakama engineers. If your problem is a crash that we've seen before — or our users have — this may get you a quick answer.

If you are certain that what you are experiencing is a bug, please open an issue on the GitHub repository. Please be as detailed as possible and outline any steps required to replicate the issue you are experiencing.

For any questions, requests or comments please contact us at support@heroiclabs.com

If you find this package useful, rate it or leave a review. It is always appreciated.