Building a Chat App in JavaScript
Best Practice Guide

Use cases based guide with PubNub APIs

- Basic Use Cases
    - Initializing the PubNub client
    - Joining a chat room
    - Joining multiple rooms
    - Leaving the room
    - Receiving messages
    - Send a message
    - Setting a unique ID (UUID) for each user
    - Restoring missed messages while a user is idling (e.g. page reload, transition between pages)
- User Presence Use Cases
    - Displaying how many users are currently online
    - Getting user online/offline status (join, leave, timeout)
    - Displaying who is online
    - Binding extra information (“State”) for each user
    - Showing an indicator when someone is typing
- Channel Grouping
    - Grouping the channels
    - Channel naming conventions
- Message History
    - Retrieving past messages
    - Displaying the messages from a certain time span
    - Retrieving more than 100 messages from history
- Secure Chat Rooms
    - Setting up a secure chat room
    - Granting a user access to a private chat room
    - Revoking the user the access (“Kick”, “Ban”)
    - Sending encrypted messages
    - Sending encrypted messages over SSL / TLS
    - Sending a message with attached digital signature
    - Receiving the messages with attached digital signature
- Push Notifications
    - Sending push notifications with Google Cloud Messenger (GCM) and Apple Push Notification Service (APNs)

Common use cases with standard HTML5 and JavaScript APIs

- Displaying the time a message is delivered
- Stripping off HTML tags from a message to prevent XSS attacks
- Tracking a user's location
- Storing small data locally in a browser

Design Patterns for common use cases that are supported by PubNub SDK

- Determining if a message is read
- Sending binary contents (photo, video, etc.)
- Removing messages from history

Overview

The purpose of this guide is to provide developer-friendly instruction on building a chat app using PubNub's JavaScript SDK. It includes use case-based code samples, recommendations on best practices, and common design patterns.

Some APIs used in this documentation are add-on features, which need to be enabled via the Developer Admin Dashboard at **admin.pubnub.com**.

Basic use cases

Initializing the PubNub client

To get started, you must register your app on your admin portal page (https://admin.pubnub.com) to obtain your `subscribe_key` and `publish_key`.

Then in your JavaScript file, include the PubNub JavaScript SDK and initialize the client:

Include `pubnub.js` in your html file. Be sure to include the latest version, which is available in the JavaScript documentation.

```
<script src="http://cdn.pubnub.com/pubnub-[version number].js"></script>
```

The JavaScript file should looks like *pubnub-3.7.14.js* (or *pubnub-3.7.14.min.js* if you wish the minified script.)

Next, in your JavaScript file, instantiate a new `Pubnub` instance:

```
var pubnub = PUBNUB.init({
      subscribe_key : 'sub-c-f762fb78-...',
      publish_key   : 'pub-c-156a6d5f-...'
});
```

Joining a chat room

"Joining" a chat room can be done by using `subscribe()` to listen on a channel, "room-1":

```
pubnub.subscribe({
      channel : 'room-1'
});
```

You can easily create a private 1:1 chat by generating a unique arbitrary channel name (e.g. 'private-36258a4') and sharing it with only 2 people to avoid the room name being easily duplicated by accident.

Joining multiple rooms

** You need to enable Stream Controller from your Admin Dashboard. (https://admin.pubnub.com)*

Subscribe to multiple channels, such as "room-1", "room-2"... on a single connection ("Multiplexing").

With channel multiplexing, a client can subscribe to many channels via a single socket connection. The PubNub Data Stream Network will manage the list on the server so that the client will receive messages from all subscribed channels.

```
pubnub.subscribe({
      channel : 'room-1, room-2, room-3'
});
```

Leaving the room

Use `unsubscribe()` to leave the channel:

```
pubnub.unsubscribe({
      channel : 'room-1'
});
```

Receiving messages

Subscribing is an async operation- upon the callback of `subscribe()`, the client will receive all messages that have been published to the channel. The `subscribe()` method is called every time a set of data is published:

```
pubnub.subscribe({
      channel  : 'room-1',
      callback : function(m) {console.log(m);}
});
```

where `m` is an object that contains the message you received.

Send a message

Using `publish()` allows you to send messages as an object:

```
pubnub.publish({
      channel : 'room-1',
      message : {
            avatar : 'grumpy-cat.jpg',
            text   : 'Hello, hoomans!'
      },
```

```
      callback : function(m) {console.log(m)}, // success callback
      error    : function(e) {console.log(e)}  // error callback
});
```

You can send any data (a string, JSON, or an object), as long as the data size is below 32k.

Setting a unique ID (UUID) for each user

When an initialization is established, a UUID is automatically generated at `init` when you initialize with PubNub by default. Alternatively, you can create a unique ID for the user and include it as a uuid.

You can generate one manually either with `PUBNUB.uuid()` method, or your own method.

The uuid can also be a unique string. For example, if your login method is a social login, such as Twitter OAuth, you can assign the username as `uuid`.

```
// generates a random uuid, or use your own method to create a unique id
var userId = PUBNUB.uuid();

var pubnub = PUBNUB.init({
      subscribe_key : 'sub-c-f762fb78-...',
      publish_key   : 'pub-c-156a6d5f-...',
      uuid          : userId
});
```

For more details about setting UUID see PubNub Knowledge Base: [How do I set a UUID](#).

Restoring missed messages while a user is idling (e.g. page reload, transition between pages)

There are times when clients may abruptly exit or disconnect. The `restore` setting enables the subscribed client to reconnect with the last messages timetoken.

```
pubnub.subscribe({
      channel : 'room-1',
      restore : true
});
```

This way, the user will reconnect with the most recently used timetoken and will continue to receive messages from that time forward.

User Presence Use Cases

Displaying how many users are currently online

** You need to enable Presence from the Admin portal.*

Use the `presence` callback for subscribe to receive `presence` events, which includes `occupancy`, and state change such as 'leave', 'join', 'timeout', and 'state-change'. (See "*Binding extra info ("State") for each user*").

```
pubnub.subscribe({
      channel  : 'room-1',
      callback : function(m) {console.log(m);},
      presence : function(m) {console.log(m.occupancy);}
});
```

where `m.occupancy` returns the number of users currently in the chat channel "room-1".

Getting user online/offline status (join, leave, timeout)

** You need to enable Presence from your Admin portal.*

```
pubnub.subscribe({
      channel  : 'room-1',
      callback : function(m) {console.log(m);},
      presence : function(m) {console.log(m.action);}
});
```

where `m.action` returns either 'join' (when a user subscribes to the channel), 'leave' (when the user unsubscribes from the channel), 'timeout' (when a user is disconnected). Also, 'state-change' is triggered when the user's state data has been modified (see *Binding extra info ("State") for each user*)

Displaying who is online

** You need to enable Presence from your Admin portal.*

You can obtain information about the current state of a channel, including a list of unique user-ids currently subscribed to the channel, by calling the `here_now()` function after the connection is established:

```
pubnub.here_now({
      channel  : 'room-1',
      callback : function(m) {console.log(m.uuid)}
});
```

The response includes `m.uuid` as an array of `uuids` as follows:

```
uuids: [
  'grumpy-cat',
  'maru',
  'lil-bab'
]
```

Binding extra information ("State") for each user

You can *set* `state`, extra information for a user, during a subscribe with the state parameter. Simply specify any desired key/value pairs as a JSON object.

```
pubnub.state({
    channel : 'room-1',
    state   : { mood : 'grumpy' }
});
```

You can *get* the state info of a user at any time, too:

```
pubnub.state({
    channel  : 'room-1',
    uuid     : 'grumpy-cat',
    callback : function(m) {console.log(m)} // returns an object containing
key/value pairs.
});
```

The `state` API allows you to provide user state data, and the data is sent to other subscribers upon a 'join' event.

The `state` can be an object of key/value pairs with any arbitrary data, as long as the data types are int, float, and string.

Showing an indicator when someone is typing

You can achieve this via `state` change.

You may want to listen to DOM events like `keydown` or `keypress` for a chat text field, and when the event is triggered, change the user's `state` to something like `{isTyping: true}`, and notify all other subscribers, as shown in the previous example, "*Binding extra info ("State") for each user*".

~~A `state-change` event is a part of Presence event and it will be fired anytime the state is changed. Please see "User Presence Use Cases".~~

Channel Grouping

Grouping the channels

** You need to enable Stream Controller from your Admin portal.*

Early we reviewed how to join multiple room over a single socket connection by using a comma delimited list. However, managing large lists can be complex. Therefore you can use Channel Groups to group up to 2,000 channels under a single Channel group.

```
pubnub.channel_group_add_channel({
      channel       : 'room-1',
      channel_group : 'Group-Cats'
      callback      : function(m){console.log(m)}
});
```

To remove a channel from the group, use `channel_group_remove_channel().`

To subscribe to a channel group:

```
pubnub.subscribe({
```

```
      channel_group : 'Group-Cats',
      callback      : function(m){console.log(m)}
});
```

For more information on grouping, please refer to our Stream Controller API documentation: http://www.pubnub.com/docs/javascript/overview/stream-controller.html

Channel naming conventions

Each channel group and associated channel(s) is identified by a unique name. These names may be a string of up to 64 Unicode characters, excluding reserved characters: , , : , . , / , *, \, non-printable ASCII control characters, and Unicode zero.

For example, the channel group, “Group-Cats” might contain the following channel names: “general”, “lolcats-1”, “lolcats-2”, “maru”, “nyancat-original”, “nyancat-variants”, etc.

Although in this example, the group name is capitalized and channel names are all in lowercase, you are free to name channels whatever you want to as long as you stick with your own consistent naming convention.

Message History

Retrieving past messages

** You need to enable Storage & Playback from your Admin portal.*

You can fetch the historical messages have been sent to a channel using `history()`.

```
pubnub.history({
      channel  : 'room-1',
      count    : 50,
      callback : function(m) { console.log(m[0]) }
});
```

where `m[0]` outputs an array of 50 most recently sent messages on the given channel, ‘room-1’. By default, last 100 messages are returned if you don’t specify a count.

100 is the maximum number of messages you can fetch by the API. If you need more than 100, see the section, **Retrieving more than 100 messages from history.**

Displaying the messages from a certain time span

** You need to enable Storage & Playback from your Admin portal.*

To retrieve messages within certain time tokens, use `start` and `end` arguments with `history()`:

```
pubnub.history({
      channel  : 'room-1',
      callback : function(m){ console.log(m) },
      start    : 14219808508935165,
      end      : 14219808515130421
});
```

The time tokens must be in valid 17 digit tokens. To convert date objects to the specific time tokens, get `dateObj.getTime()` and multiply by 10000:

```
var fiveMinAgo = (new Date().getTime() - (5*60*1000)) * 10000;
```

Retrieving more than 100 messages from history

By default, history returns maximum 100 messages. To retrieve more than 100, use timestamp to page though the history.

```
getAllMessages = function(timetoken) {
    pubnub.history({
        start: timetoken,
        channel: channel,
        callback: function(payload) {
            var msgs = payload[0];
            var start = payload[1];
            var end = payload[2];

            if (msgs.length === 100) getAllMessages(start);
        }
    });
}
```

Or you can use the wrapper, *pubnub-flex-history* to handles it automatically: https://github.com/pubnub/pubnub-flex-history

Secure Chat Rooms

PubNub Access Manager allows you to manage granular permissions for your realtime apps and data. For example, you can use the Access Manager to create a secure private channel for a subset of users only.

Setting up a secure chat room

** You need to enable Access Manager from your Admin portal.*

Note: if you are using PubNub JavaScript SDK version 3.6 or older, you need to include

```
<script src="https://cdn.pubnub.com/pubnub-crypto.min.js"></script>
```

Using Access Manager, you can set admin mode for who has access to grant/deny a user permission, and general user mode.

Admin Mode

First, after you enable Access Manager on your PubNub Admin Dashboard, you need to obtain your **`secret_key`**, along with your subscribe and publish keys.

Then create an instance with Admin Granting Capabilities by including the `secret_key`.
All the admin actions (setting `secret_key`, `grant`, `revoke`, etc.) should be done on the server-side, since you don't want to expose the `secret_key` on the client side.

The example below is in node.js:

```
var pubnub = require('pubnub').init({
    subscribe_key : 'sub-c-f762fb78-...',
    publish_key   : 'pub-c-156a6d5f-...',
    secret_key    : 'sec-c-M2U0N...'
});
```

User Mode

Initialize your client with `auth_key` to identify each subscriber.:

```
var pubnub = PUBNUB.init({
     subscribe_key : 'sub-c-f762fb78-...',
     publish_key   : 'pub-c-156a6d5f-...',
     auth_key      : 'my_auth_key_from_login'
});
```

You should generate an arbitrary `auth_key`. It can be an `uuid`, or an authentication token from OAuth, Facebook Connect or any other authentication service. Then the admin grants access to users using the `auth_key`.

To reset the `auth_key, use auth()` method:

```
pubnub.auth('my_new_auth_key');
```

Granting a user access to a private chat room

** You need to enable Access Manager from your Admin portal.*

Only admin can grant access to a user. **This operation should be done on the server-side**.

This example grants a user. who is associated with the particular auth_key, 60 minute read and write privilege on a particular channel:

```
pubnub.grant({
    channel  : 'private-83659357',
    auth_key : 'abcxyz123-auth-key',
    read     : true,
    write    : true,
    ttl      : 60,
    callback : function(m){console.log(m)}
});
```

The default `ttl` value is 24 hours (1440 minutes).

If the auth_key param is not included, a global grant is issued to all users who are subscribed to the channel.

Revoking the user the access (“Kick”, “Ban”)

Only admin can revoke access to a user. **This operation should be done on the server-side**.

```
pubnub.revoke({
    channel  : 'private-83659357',
    auth_key : 'abcxyz123-auth-key',
    ttl      : 60
});
```

If the auth_key param is not included, access is revoked globally for all users who are subscribed to the channel.

Sending encrypted messages

PubNub client libraries offer built-in Advanced Encryption Standard (AES) 256-bit encryption. To use message encryption, set a `cipher_key`. Only a recipient who has the cipher key is able to read the data, so the key needs to be hidden from any users, thus this is generally a good idea to keep it on server side.

Then publish/subscribe as usual.

```
var pubnub = PUBNUB.init({
     subscribe_key : 'sub-c-f762fb78-...',
     publish_key   : 'pub-c-156a6d5f-...',
     cipher_key    : 'my_cipher_key'
});
```

For more details, read:
https://www.pubnub.com/docs/javascript/tutorial/message-encryption.html

Sending encrypted messages over SSL / TLS

Enable Transport Layer Encryption with SSL/TLS by setting `ssl` param to `true`.
Then publish/subscribe as usual.

```
var pubnub = PUBNUB.init({
     subscribe_key : 'sub-c-f762fb78-...',
     publish_key   : 'pub-c-156a6d5f-...',
     ssl           : true
});
```

For more details, read:
https://www.pubnub.com/docs/javascript/tutorial/transport-layer-security.html

Sending a message with attached digital signature

Malicious users may attempt to spoof the identity of users to impersonate other users. You may consider using a digital signature as a fingerprint.

To attach a digital signature to a message, first, create the signature by concatenation of the following base string (username + message). Then, calculate a message digest using SHA1 (signature_base_string) or SHA256 (signature_base_string).

Now you can use this cryptographically generated hash to sign with the users' private key using an ECC/RSA asymmetric algorithm.

Then attach it with the message object to publish:

```
{
    username  : "Grumpy Cat",
    message   : "Hello hoomans",
    signature : "tnnArxj06cWHq44gCs1OSKk/jLY"
}
```

Receiving the messages with attached digital signature

To receive the message with the signature, you will use the signature to verify the authenticity of the message. If the message was an attempted spoof, you will drop the message.

To verify the signature, recreate the signature by concatenation of the base string (username + message), then calculate a message digest using SHA1 or SHA256. Compare with the original message digest.

It is important to note that you should only print the username retrieved from the Public Key Trusted Source and not from the message content. The message content is only used for signing and verifying Identity. If an Identity is changed it must be relayed via the Public Key Infrastructure (PKI) using PubNub's Broadcast mechanism and Storage and Playback for Secure Authorization.

Access tokens, Identity Name/Photo and Asymmetric Keys must only be generated from trusted execution environments on your servers and relayed over a PubNub Authorized Read-only PKI data channel.

For more info, see
http://www.pubnub.com/blog/chat-security-user-identification-with-digital-signature-message-verification/

Push Notifications

Sending push notifications with Google Cloud Messenger (GCM) and Apple Push Notification Service (APNs)

** You need to enable Push Notifications from your Admin Dashboard.*

Push Notifications are used when the user closes the chat application or runs it in the background. On Apple devices the only solution is to use Apple Push Notification Service (APNs) essentially PubNub will pass you messages to Apple and they will deliver it to the phone to alert users. On Android you have 2 choices. 1) Use Google Cloud Messaging (GCM) similar to APNs where PubNub passes the messages off to Google and they deliver the messages. 2) Use a Background Thread this allows you to have the same low-latency message delivery. (See more on using Android background thread.

First, you must be registered to use Google Cloud Messaging (GCM) for Android, or Apple Push Notification Service (APNs) for iOS devices.

Each device that runs your app has a unique identifier called, either *registration ID* or *device token*, depending on its OS, and you need to register the token from your user's device to be able to send push notifications.

You need to code in either your native app, or use Cordova/PhoneGap with the Cordova Push plugin (https://github.com/phonegap-build/PushPlugin) to obtain the token from your user.

For example, Android device ID size is up to 4k, and the string starts with "APA", and APNS token is about 32 bytes in size.

Once you get the device-specific ID, save the token, then use PubNub `mobile_gw_provision` method and `PNmessage` object to set GCM or APNs message objects, to finally send a push message to the specific device.

```
var channel = 'room-1';

pubnub.mobile_gw_provision({
    device_id : regid, // device-specific ID
    channel   : channel,
    op        : 'add', // or ‘remove’
    gw_type   : 'apns', // or 'gcm' for Android
    callback  : function() {
        var message      = PNmessage();
        message.pubnub   = pubnub;
        message.callback = function (msg){ console.log(msg); };
        message.error    = function (msg){ console.log(msg); };
        message.channel  = channel;

        message.apns     = {
            alert : 'Yo! from GrumpyCat'
        };

        message.gcm = {
            title   : 'PubNub Chatterbox',
            message : 'Yo! from GrumpyCat'
        };

        message.publish();
    }
});
```

For more information, please read tutorials:
http://www.pubnub.com/blog/sending-android-push-notifications-via-gcm-javascript-using-phonegap/ and
http://www.pubnub.com/blog/sending-ios-push-notifications-via-apns-javascript-using-apns-phonegap/

Common use cases with standard HTML5 and JavaScript APIs

Displaying the time a message is delivered

To display the time token in human-readable form, use JavaScript `Date` object methods.

Using the subscribe success callback, the e returns an envelope that contains a 17-digit precision unix time (UTC), upon a publish success the callback returns the following:

```
p.subscribe({
      channel  : channel,
      callback : function(m, e, c) {
          var t = e[1];
      }
});
```

To convert the timestamp to a UNIX timestamp, divide it by 10,000,000. For example, to obtain a locale date string in JavaScript,

```
var unixTime = t * 10000000;
var d = new Date(unixTime * 1000);
var localeDateTime = d.toLocaleString();
```

or directly from 17-digit precision time without converting to UNIX time,

```
var d = new Date(t/10000);
var localeDateTime = d.toLocaleString();
```

where localDateTime is something similar to `"7/5/2015, 3:58:43 PM"`

Stripping off HTML tags from a message to prevent XSS attacks

User input from text fields may contain HTML tags, and especially `<script>` injections can be malicious. You should validate user inputs properly to avoid this.

The easiest way to prevent this is to simply remove <> to make the HTML into a string.

```
var text      = input.value();
```

```
var safe_text = ('' + text).replace(/[<>]/g, '');
```

Alternatively, you may want to to escape the special characters, rather than just stripping off. See OWASP's XSS prevention cheat sheet for more information.

Tracking a user's location

You may want to include users' physical locations as the user info.

Use HTML5 geolocation API to get the user's current location by using either WiFi, cell tower, GPS or AGPS from the browser:

```
if ("geolocation" in navigator) { // feature detection
  navigator.geolocation.getCurrentPosition(function(position) {
      var lat = position.coords.latitude;
      var lon = position.coords.longitude;
  });
}
```

For example, the location of PubNub office (in San Francisco) returns as `lat` = 37.7833866, and `lon` = -122.3995498.

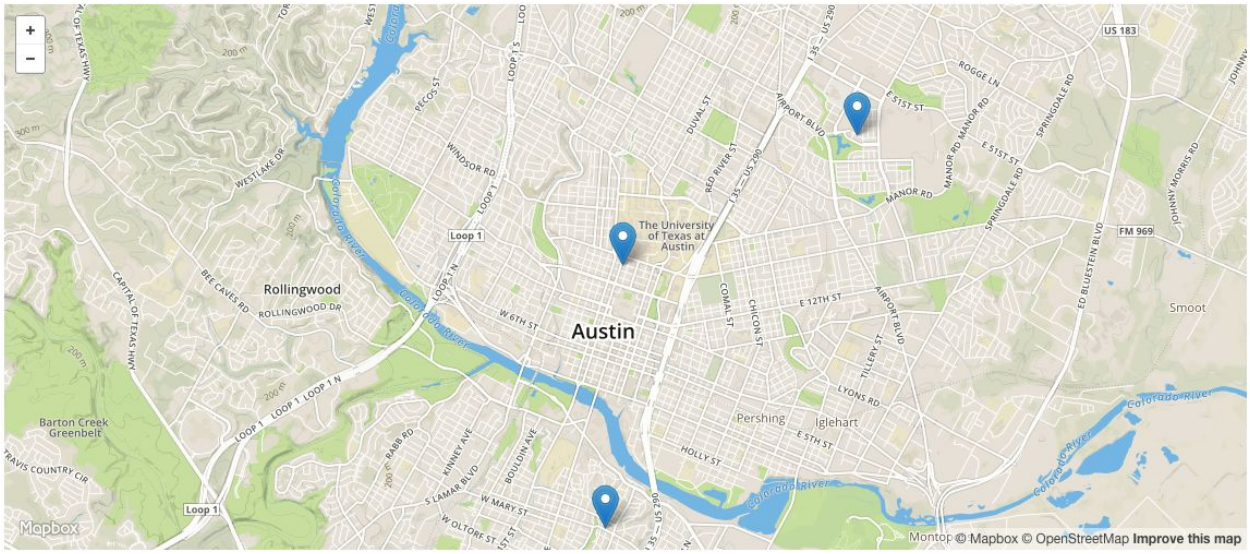
Placing a marker of the user location on a map

~~To plot static locations using the latitude and longitude using a 3rd party map API:~~

To plot realtime locations, you can use PubNub opensource EON libraries to track location data from a PubNub stream and plot on Mapbox map:
http://www.pubnub.com/developers/eon/map/extra/

~~You can also use the 3rd party mapping APIs such as:~~

- ~~MapBox Marker API:~~
  ~~https://www.mapbox.com/mapbox.js/example/v1.0.0/l-mapbox-marker/~~
- ~~Google Maps Maker API:~~
  ~~https://developers.google.com/maps/documentation/javascript/examples/marker-simple~~



Obtaining a meaningful place name from the user location

To get the city names, use the 3rd party APIs such as Geocoding API provided by Mapbox:

With Mapbox Reverse Goecoding API the requested feature data located at the input `{lon},{lat}` coordinates returns with a response includes at most one result from each geocoding dataset.

The example request:
`https://api.mapbox.com/v4/geocode/mapbox.places/-122.3994321,37.783544.json?access_token=[your_mapbox_access_token]`
and its response:

```
{"type":"FeatureCollection","query":[-122.3994321,37.783544],"features":[{"
id":"address.275745210758781","type":"Feature","text":"3rd
St","place_name":"300 3rd St, San Francisco, 94103, California, United
States","relevance":1, ...
```

Storing small data locally in a browser

Storing small data locally on a user's browser is easier, and can be a more suitable option than using an external database to store the data. For example, tracking if the user has logged in

from a particular browser. You can do so with W3C Web Storage API using `sessionStorage` (non persistent) or `localStorage` (persistent).

To set data item:

```
localStorage.setItem('username', 'Nyan Cat');
```

To get the data:

```
var username = localStorage.getItem('username');
```

Web Storage is easier than storing data in cookies and does not share the data with the server.

Design Patterns for the features not directly supported by PubNub APIs

Determining if a message is read

There are many different design patterns, depending on your specific user interactions (e.g. click a message to read), to determine if a message has been read.

One general idea is to simply keep track of the PubNub timetoken to mark the "last read" messages, while the user is online.

See the section, **Getting user online/offline status** to see how to determine a user is online or not.

Timetoken can be retrieved at subscribe callback. (See the section **Displaying the time a message is delivered**)

Alternatively, you track the “read” messages by moving to another channel, such as "channel-READ" and check the IDs of the messages you want to mark as `read` or `unread`.

Sending binary contents (photo, video, etc.)

The publish API allows you to send messages up to 32kb, so if the file is small enough, encode a binary file in Base64 format and publish the data as usual.

To achieve this in JavaScript, first you need to convert an image to a canvas object then use `toDataURL` method.

```
var dataURL = canvas.toDataURL('image/jpeg');
```

Alternatively, use *FileReader API* to encode an image a base64, with file uploader, `<input type=”file>` with `readAsDataURL` method.

Or you can use the 3rd party modules to convert binary to base64, if you are coding in node.js.

When the file size is above 32kb, you need to provide your own storage servers.
You can store the binary files in a database such as Amazon S3, or other cloud storage services or on a CDN then send the URL over PubNub.

Removing messages from history

PubNub stores all messages per channel with the *Storage & Playback* feature add-on. Since it stores all messages in a time series, and doesn’t expose Updating and Deleting, you need to implement this functionality.

There are two ways to do this:

- Interleaving
- Side-channel

Both cases, you send a message including `deleted` data with the default value of `false.`

```
{
      message_id : 10001,
      user       : “GrumpyCat”,
      deleted    : false
}
```

With the Interleaving pattern, you publish new versions of the same message to the same channel just like normal publishes. The subsequent messages must use the same `message_id` as the original message, and to simplify rendering, the messages for updates should contain the full content so that the original is not needed.

In the Side Channel pattern, you are publishing any updates and deletes to a separate channel that you also subscribe to. This means that you have a primary content channel and a side channel for updates/deletes.
With the Storage & Playback feature add-on, messages are stored on a per channel basis. This means that the side-channel has only update/delete messages in it, rather than both the original messages and the changes.

For the full explanations, see:
http://pubnub.github.io/pubnub-design-patterns/2015/02/26/Message-Update-Delete.html

---

References: Currently Available Articles

Chat App Demos and Blogs

- Solutions/chat: http://www.pubnub.com/solutions/chat/
- Angular Chat: http://www.pubnub.com/developers/angularjs/
- Blog: http://www.pubnub.com/blog/tag/chat/

Technical Blog Articles

- 10 Chat: http://www.pubnub.com/developers/demos/10chat/
- WebRTC Video Chat: http://www.pubnub.com/blog/building-a-webrtc-video-and-voice-chat-application/ and http://www.pubnub.com/blog/6-essentials-for-webrtc-video-and-voice-app-dev/
- Chat User Identification with Digital Signature Message Verification https://gist.github.com/stephenlb/266cd8beecf3caf0f629
- Multiplex with Ember.js: http://www.pubnub.com/blog/combining-chat-data-streams-emberjs-multiplexing/
- Access management with Ember.js: http://www.pubnub.com/blog/implementing-access-management-ember-js-messaging/

- Cipher key: http://www.pubnub.com/blog/ember-js-encryption-and-three-way-data-binding-for-chat/
- Polymer <pubnub-element>: http://www.pubnub.com/blog/creating-a-polymer-chat-app-with-material-design/
- MemeWarz (10 articles): http://www.pubnub.com/blog/multiplayer-game-lobby-with-pubnub/
- Babel (4 articles): http://www.pubnub.com/blog/self-destructing-chat-key-exchange-self-destructing-messages/
- Geohash: http://www.pubnub.com/blog/geohashing-chat-by-proximity/
- Access Manager: http://www.pubnub.com/blog/secure-chat-using-pubnub-access-manager/
- iOS (Obj-C): http://www.pubnub.com/blog/building-an-ios-group-messenger-app/
- Group chat (3 articles): http://www.pubnub.com/blog/ditch-your-backend-with-pubnub-messenger/
- 10 lines: http://www.pubnub.com/blog/build-real-time-chat-10-lines-code/