Baileys - Typescript/Javascript WhatsApp Web API

Important Note

This library was originally a project for **CS-2362 at Ashoka University** and is in no way affiliated with or endorsed by WhatsApp. Use at your own discretion. Do not spam people with this. We discourage any stalkerware, bulk or automated messaging usage.

Liability and License Notice

Baileys and its maintainers cannot be held liable for misuse of this application, as stated in the MIT license. The maintainers of Baileys do not in any way condone the use of this application in practices that violate the Terms of Service of WhatsApp. The maintainers of this application call upon the personal responsibility of its users to use this application in a fair way, as it is intended to be used. ##

Baileys does not require Selenium or any other browser to be interface with WhatsApp Web, it does so directly using a **WebSocket**. Not running Selenium or Chromimum saves you like **half a gig** of ram :/ Baileys supports interacting with the multi-device & web versions of WhatsApp. Thank you to @pokearaujo for writing his observations on the workings of WhatsApp Multi-Device. Also, thank you to @Sigalor for writing his observations on the workings of WhatsApp Web and thanks to @Rhymen for the **go** implementation.

Please Read

The original repository had to be removed by the original author - we now continue development in this repository here. This is the only official repository and is maintained by the community. **Join the Discord here**

Example

Do check out & run example.ts to see an example usage of the library. The script covers most common use cases. To run the example script, download or clone the repo and then type the following in a terminal:

1. `cd path/to/Baileys`
2. `yarn`
3. `yarn example`

Install

Use the stable version:

```
yarn add @whiskeysockets/baileys
```

Use the edge version (no guarantee of stability, but latest fixes + features)

```
yarn add github:WhiskeySockets/Baileys
```

Then import your code using:

```
import makeWASocket from '@whiskeysockets/baileys'
```

Unit Tests

TODO

Connecting multi device (recommended)

WhatsApp provides a multi-device API that allows Baileys to be authenticated as a second WhatsApp client by scanning a QR code with WhatsApp on your phone.

```
import makeWASocket, { DisconnectReason } from '@whiskeysockets/baileys'
import { Boom } from '@hapi/boom'

async function connectToWhatsApp () {
    const sock = makeWASocket({
        // can provide additional config here
        printQRInTerminal: true
    })
    sock.ev.on('connection.update', (update) => {
        const { connection, lastDisconnect } = update
        if(connection === 'close') {
            const shouldReconnect = (lastDisconnect.error as Boom)?.output?.statusCode !== DisconnectReason.loggedOut
            console.log('connection closed due to ', lastDisconnect.error, ', reconnecting ', shouldReconnect)
            // reconnect if not logged out
            if(shouldReconnect) {
                connectToWhatsApp()
            }
        } else if(connection === 'open') {
            console.log('opened connection')
        }
    })
    sock.ev.on('messages.upsert', m => {
        console.log(JSON.stringify(m, undefined, 2))

        console.log('replying to', m.messages[0].key.remoteJid)
        await sock.sendMessage(m.messages[0].key.remoteJid!, { text: 'Hello there!' })
    })
}
// run in main file
connectToWhatsApp()
```

If the connection is successful, you will see a QR code printed on your terminal screen, scan it with WhatsApp on your phone and you'll be logged in!

Configuring the Connection

You can configure the connection by passing a `SocketConfig` object.

The entire `SocketConfig` structure is mentioned here with default values:

```
type SocketConfig = {
    /** the WS url to connect to WA */
    waWebSocketUrl: string | URL
    /** Fails the connection if the socket times out in this interval */
    connectTimeoutMs: number
    /** Default timeout for queries, undefined for no timeout */
    defaultQueryTimeoutMs: number | undefined
    /** ping-pong interval for WS connection */
    keepAliveIntervalMs: number
    /** proxy agent */
    agent?: Agent
    /** pino logger */
    logger: Logger
    /** version to connect with */
    version: WAVersion
    /** override browser config */
    browser: WABrowserDescription
    /** agent used for fetch requests -- uploading/downloading media */
    fetchAgent?: Agent
    /** should the QR be printed in the terminal */
    printQRInTerminal: boolean
    /** should events be emitted for actions done by this socket connection */
    emitOwnEvents: boolean
    /** provide a cache to store media, so does not have to be re-uploaded */
    mediaCache?: NodeCache
    /** custom upload hosts to upload media to */
    customUploadHosts: MediaConnInfo['hosts']
    /** time to wait between sending new retry requests */
    retryRequestDelayMs: number
    /** max msg retry count */
    maxMsgRetryCount: number
    /** time to wait for the generation of the next QR in ms */
    qrTimeout?: number;
    /** provide an auth state object to maintain the auth state */
    auth: AuthenticationState
    /** manage history processing with this control; by default will sync up everything */
    shouldSyncHistoryMessage: (msg: proto.Message.IHistorySyncNotification) => boolean
    /** transaction capability options for SignalKeyStore */
    transactionOpts: TransactionCapabilityOptions
    /** provide a cache to store a user's device list */
    userDevicesCache?: NodeCache
    /** marks the client as online whenever the socket successfully connects */
    markOnlineOnConnect: boolean
    /**
     * map to store the retry counts for failed messages;
     * used to determine whether to retry a message or not */
    msgRetryCounterMap?: MessageRetryMap
    /** width for link preview images */
    linkPreviewImageThumbnailWidth: number
    /** Should Baileys ask the phone for full history, will be received async */
    syncFullHistory: boolean
    /** Should baileys fire init queries automatically, default true */
    fireInitQueries: boolean
    /**
     * generate a high quality link preview,
     * entails uploading the jpegThumbnail to WA
     * */
    generateHighQualityLinkPreview: boolean

    /** options for axios */
    options: AxiosRequestConfig<any>
    /**
     * fetch a message from your store
     * implement this so that messages failed to send (solves the "this message can take a while" issue) can be retried
     * */
    getMessage: (key: proto.IMessageKey) => Promise<proto.IMessage | undefined>
}
```

Emulating the Desktop app instead of the web

1. Baileys, by default, emulates a chrome web session
2. If you'd like to emulate a desktop connection (and receive more message history), add this to your Socket config:

```
const conn = makeWASocket({
    ...otherOpts,
    // can use Windows, Ubuntu here too
    browser: Browsers.macOS('Desktop'),
    syncFullHistory: true
})
```

Saving & Restoring Sessions

You obviously don't want to keep scanning the QR code every time you want to connect.

So, you can load the credentials to log back in:

```
import makeWASocket, { BufferJSON, useMultiFileAuthState } from '@whiskeysockets/baileys'
import * as fs from 'fs'

// utility function to help save the auth state in a single folder
// this function serves as a good guide to help write auth & key states for SQL/no-SQL databases, which I would recommend in any pro
const { state, saveCreds } = await useMultiFileAuthState('auth_info_baileys')
// will use the given state to connect
// so if valid credentials are available -- it'll connect without QR
const conn = makeWASocket({ auth: state })
// this will be called as soon as the credentials are updated
conn.ev.on ('creds.update', saveCreds)
```

Note: When a message is received/sent, due to signal sessions needing updating, the auth keys (`authState.keys`) will update. Whenever that happens, you must save the updated keys (`authState.keys.set()` is called). Not doing so will prevent your messages from reaching the recipient & cause other unexpected consequences. The `useMultiFileAuthState` function automatically takes care of that, but for any other serious implementation -- you will need to be very careful with the key state management.

Listening to Connection Updates

Baileys now fires the `connection.update` event to let you know something has updated in the connection. This data has the following structure:

```
type ConnectionState = {
    /** connection is now open, connecting or closed */
    connection: WAConnectionState
    /** the error that caused the connection to close */
    lastDisconnect?: {
        error: Error
        date: Date
    }
    /** is this a new login */
    isNewLogin?: boolean
    /** the current QR code */
    qr?: string
    /** has the device received all pending notifications while it was offline */
    receivedPendingNotifications?: boolean
}
```

Note: this also offers any updates to the QR

Handling Events

Baileys uses the EventEmitter syntax for events. They're all nicely typed up, so you shouldn't have any issues with an Intellisense editor like VS Code.

The events are typed as mentioned here:

```
export type BaileysEventMap = {
    /** connection state has been updated -- WS closed, opened, connecting etc. */
    'connection.update': Partial<ConnectionState>
    /** credentials updated -- some metadata, keys or something */
    'creds.update': Partial<AuthenticationCreds>
    /** history sync, everything is reverse chronologically sorted */
    'messaging-history.set': {
        chats: Chat[]
        contacts: Contact[]
        messages: WAMessage[]
        isLatest: boolean
    }
    /** upsert chats */
    'chats.upsert': Chat[]
    /** update the given chats */
    'chats.update': Partial<Chat>[]
    /** delete chats with given ID */
    'chats.delete': string[]
    'labels.association': LabelAssociation
    'labels.edit': Label
    /** presence of contact in a chat updated */
    'presence.update': { id: string, presences: { [participant: string]: PresenceData } }

    'contacts.upsert': Contact[]
    'contacts.update': Partial<Contact>[]

    'messages.delete': { keys: WAMessageKey[] } | { jid: string, all: true }
    'messages.update': WAMessageUpdate[]
    'messages.media-update': { key: WAMessageKey, media?: { ciphertext: Uint8Array, iv: Uint8Array }, error?: Boom }[]
    /**
     * add/update the given messages. If they were received while the connection was online,
     * the update will have type: "notify"
     *  */
    'messages.upsert': { messages: WAMessage[], type: MessageUpsertType }
    /** message was reacted to. If reaction was removed -- then "reaction.text" will be falsey */
    'messages.reaction': { key: WAMessageKey, reaction: proto.IReaction }[]

    'message-receipt.update': MessageUserReceiptUpdate[]

    'groups.upsert': GroupMetadata[]
    'groups.update': Partial<GroupMetadata>[]
    /** apply an action to participants in a group */
    'group-participants.update': { id: string, participants: string[], action: ParticipantAction }

    'blocklist.set': { blocklist: string[] }
    'blocklist.update': { blocklist: string[], type: 'add' | 'remove' }
    /** Receive an update on a call, including when the call was received, rejected, accepted */
    'call': WACallEvent[]
}
```

You can listen to these events like this:

```
const sock = makeWASocket()
sock.ev.on('messages.upsert', ({ messages }) => {
    console.log('got messages', messages)
})
```

Implementing a Data Store

Baileys does not come with a defacto storage for chats, contacts, or messages. However, a simple in-memory implementation has been provided. The store listens for chat updates, new messages, message updates, etc., to always have an up-to-date version of the data.

It can be used as follows:

```
import makeWASocket, { makeInMemoryStore } from '@whiskeysockets/baileys'
// the store maintains the data of the WA connection in memory
// can be written out to a file & read from it
const store = makeInMemoryStore({ })
// can be read from a file
store.readFromFile('./baileys_store.json')
// saves the state to a file every 10s
setInterval(() => {
    store.writeToFile('./baileys_store.json')
}, 10_000)

const sock = makeWASocket({ })
// will listen from this socket
// the store can listen from a new socket once the current socket outlives its lifetime
store.bind(sock.ev)

sock.ev.on('chats.upsert', () => {
    // can use "store.chats" however you want, even after the socket dies out
    // "chats" => a KeyedDB instance
    console.log('got chats', store.chats.all())
})

sock.ev.on('contacts.upsert', () => {
    console.log('got contacts', Object.values(store.contacts))
})
```

The store also provides some simple functions such as `loadMessages` that utilize the store to speed up data retrieval.

Note: I highly recommend building your own data store especially for MD connections, as storing someone's entire chat history in memory is a terrible waste of RAM.

Sending Messages

Send all types of messages with a single function:

Non-Media Messages

```
import { MessageType, MessageOptions, Mimetype } from '@whiskeysockets/baileys'

const id = 'abcd@s.whatsapp.net' // the WhatsApp ID
// send a simple text!
const sentMsg  = await sock.sendMessage(id, { text: 'oh hello there' })
// send a reply messagge
const sentMsg  = await sock.sendMessage(id, { text: 'oh hello there' }, { quoted: message })
// send a mentions message
const sentMsg  = await sock.sendMessage(id, { text: '@12345678901', mentions: ['12345678901@s.whatsapp.net'] })
// send a location!
const sentMsg  = await sock.sendMessage(
    id,
    { location: { degreesLatitude: 24.121231, degreesLongitude: 55.1121221 } }
)
// send a contact!
const vcard = 'BEGIN:VCARD\n' // metadata of the contact card
            + 'VERSION:3.0\n'
            + 'FN:Jeff Singh\n' // full name
            + 'ORG:Ashoka Uni;\n' // the organization of the contact
            + 'TEL;type=CELL;type=VOICE;waid=911234567890:+91 12345 67890\n' // WhatsApp ID + phone number
            + 'END:VCARD'
const sentMsg  = await sock.sendMessage(
    id,
    {
        contacts: {
            displayName: 'Jeff',
            contacts: [{ vcard }]
        }
    }
)

const reactionMessage = {
    react: {
        text: "□", // use an empty string to remove the reaction
        key: message.key
    }
}

const sendMsg = await sock.sendMessage(id, reactionMessage)
```

Sending messages with link previews

1. By default, WA MD does not have link generation when sent from the web
2. Baileys has a function to generate the content for these link previews
3. To enable this function's usage, add `link-preview-js` as a dependency to your project with `yarn add link-preview-js`
4. Send a link:

```
// send a link
const sentMsg  = await sock.sendMessage(id, { text: 'Hi, this was sent using https://github.com/adiwajshing/baileys' })
```

Media Messages

Sending media (video, stickers, images) is easier & more efficient than ever.

- You can specify a buffer, a local url or even a remote url.
- When specifying a media url, Baileys never loads the entire buffer into memory; it even encrypts the media as a readable stream.

```
import { MessageType, MessageOptions, Mimetype } from '@whiskeysockets/baileys'
// Sending gifs
await sock.sendMessage(
    id,
    {
        video: fs.readFileSync("Media/ma_gif.mp4"),
        caption: "hello!",
        gifPlayback: true
    }
)

await sock.sendMessage(
    id,
    {
        video: "./Media/ma_gif.mp4",
        caption: "hello!",
        gifPlayback: true,
    ptv: false // if set to true, will send as a `video note`
    }
)

// send an audio file
await sock.sendMessage(
    id,
    { audio: { url: "./Media/audio.mp3" }, mimetype: 'audio/mp4' }
    { url: "Media/audio.mp3" }, // can send mp3, mp4, & ogg
)
```

Notes

- `id` is the WhatsApp ID of the person or group you're sending the message to.
  - It must be in the format `[country code][phone number]@s.whatsapp.net`
    - Example for people: `+19999999999@s.whatsapp.net`.
    - For groups, it must be in the format `123456789-123345@g.us`.
  - For broadcast lists, it's `[timestamp of creation]@broadcast`.
  - For stories, the ID is `status@broadcast`.
- For media messages, the thumbnail can be generated automatically for images & stickers provided you add `jimp` or `sharp` as a dependency in your project using `yarn add jimp` or `yarn add sharp`. Thumbnails for videos can also be generated automatically, though, you need to have `ffmpeg` installed on your system.
- **MiscGenerationOptions**: some extra info about the message. It can have the following **optional** values:

  ```
  const info: MessageOptions = {
      quoted: quotedMessage, // the message you want to quote
      contextInfo: { forwardingScore: 2, isForwarded: true }, // some random context info (can show a forwarded message with thi
      timestamp: Date(), // optional, if you want to manually set the timestamp of the message
      caption: "hello there!", // (for media messages) the caption to send with the media (cannot be sent with stickers though)
      jpegThumbnail: "23GD#4/==", /*  (for location & media messages) has to be a base 64 encoded JPEG if you want to send a cus
                                      or set to null if you don't want to send a thumbnail.
                                      Do not enter this field if you want to automatically generate a thumb
                                  */
      mimetype: Mimetype.pdf, /* (for media messages) specify the type of media (optional for all media types except documents),
                                      import {Mimetype} from '@whiskeysockets/baileys'
                                  */
      fileName: 'somefile.pdf', // (for media messages) file name for the media
      /* will send audio messages as voice notes, if set to true */
      ptt: true,
      /** Should it send as a disappearing messages.
       * By default 'chat' -- which follows the setting of the chat */
      ephemeralExpiration: WA_DEFAULT_EPHEMERAL
  }
  ```

Forwarding Messages

```
const msg = getMessageFromStore('455@s.whatsapp.net', 'HSJHJWH7323HSJSJ') // implement this on your end
await sock.sendMessage('1234@s.whatsapp.net', { forward: msg }) // WA forward the message!
```

Reading Messages

A set of message keys must be explicitly marked read now. In multi-device, you cannot mark an entire "chat" read as it were with Baileys Web. This means you have to keep track of unread messages.

```
const key = {
    remoteJid: '1234-123@g.us',
    id: 'AHASHH123123AHGA', // id of the message you want to read
    participant: '912121232@s.whatsapp.net' // the ID of the user that sent the  message (undefined for individual chats)
}
// pass to readMessages function
// can pass multiple keys to read multiple messages as well
await sock.readMessages([key])
```

The message ID is the unique identifier of the message that you are marking as read. On a `WAMessage`, the `messageID` can be accessed using `messageID = message.key.id`.

Update Presence

```
await sock.sendPresenceUpdate('available', id)
```

This lets the person/group with `id` know whether you're online, offline, typing etc.

`presence` can be one of the following:

```
type WAPresence = 'unavailable' | 'available' | 'composing' | 'recording' | 'paused'
```

The presence expires after about 10 seconds.

Note: In the multi-device version of WhatsApp -- if a desktop client is active, WA doesn't send push notifications to the device. If you would like to receive said notifications -- mark your Baileys client offline using `sock.sendPresenceUpdate('unavailable')`

Downloading Media Messages

If you want to save the media you received

```
import { writeFile } from 'fs/promises'
import { downloadMediaMessage } from '@whiskeysockets/baileys'

sock.ev.on('messages.upsert', async ({ messages }) => {
    const m = messages[0]

    if (!m.message) return // if there is no text or media message
    const messageType = Object.keys (m.message)[0]// get what type of message it is -- text, image, video
    // if the message is an image
    if (messageType === 'imageMessage') {
        // download the message
        const buffer = await downloadMediaMessage(
            m,
            'buffer',
            { },
            {
                logger,
                // pass this so that baileys can request a reupload of media
                // that has been deleted
                reuploadRequest: sock.updateMediaMessage
            }
        )
        // save to file
        await writeFile('./my-download.jpeg', buffer)
    }
}
```

Note: WhatsApp automatically removes old media from their servers. For the device to access said media -- a re-upload is required by another device that has it. This can be accomplished using:

```
const updatedMediaMsg = await sock.updateMediaMessage(msg)
```

Deleting Messages

```
 const jid = '1234@s.whatsapp.net' // can also be a group
const response = await sock.sendMessage(jid, { text: 'hello!' }) // send a message
// sends a message to delete the given message
// this deletes the message for everyone
await sock.sendMessage(jid, { delete: response.key })
```

Note: deleting for oneself is supported via `chatModify` (next section)

Updating Messages

```
 const jid = '1234@s.whatsapp.net'

await sock.sendMessage(jid, {
      text: 'updated text goes here',
      edit: response.key,
    });
```

Modifying Chats

WA uses an encrypted form of communication to send chat/app updates. This has been implemented mostly and you can send the following updates:

- Archive a chat

  ```
   const lastMsgInChat = await getLastMessageInChat('123456@s.whatsapp.net') // implement this on your end
  await sock.chatModify({ archive: true, lastMessages: [lastMsgInChat] }, '123456@s.whatsapp.net')
  ```

- Mute/unmute a chat

  ```
   // mute for 8 hours
  await sock.chatModify({ mute: 8*60*60*1000 }, '123456@s.whatsapp.net', [])
  // unmute
  await sock.chatModify({ mute: null }, '123456@s.whatsapp.net', [])
  ```

- Mark a chat read/unread

  ```
   const lastMsgInChat = await getLastMessageInChat('123456@s.whatsapp.net') // implement this on your end
  // mark it unread
  await sock.chatModify({ markRead: false, lastMessages: [lastMsgInChat] }, '123456@s.whatsapp.net')
  ```

- Delete a message for me

  ```
   await sock.chatModify(
    { clear: { messages: [{ id: 'ATWYHDNNWU81732J', fromMe: true, timestamp: "1654823909" }] } },
    '123456@s.whatsapp.net',
    []
    )
  ```

- Delete a chat

  ```
   const lastMsgInChat = await getLastMessageInChat('123456@s.whatsapp.net') // implement this on your end
  await sock.chatModify({
    delete: true,
    lastMessages: [{ key: lastMsgInChat.key, messageTimestamp: lastMsgInChat.messageTimestamp }]
  },
  '123456@s.whatsapp.net')
  ```

- Pin/unpin a chat

```
 await sock.chatModify({
   pin: true // or `false` to unpin
 },
 '123456@s.whatsapp.net')
```

- Star/unstar a message

```
 await sock.chatModify({
 star: {
     messages: [{ id: 'messageID', fromMe: true // or `false` }],
         star: true // - true: Star Message; false: Unstar Message
 }},'123456@s.whatsapp.net');
```

Note: if you mess up one of your updates, WA can log you out of all your devices and you'll have to log in again.

Disappearing Messages

```
const jid = '1234@s.whatsapp.net' // can also be a group
// turn on disappearing messages
await sock.sendMessage(
    jid,
    // this is 1 week in seconds -- how long you want messages to appear for
    { disappearingMessagesInChat: WA_DEFAULT_EPHEMERAL }
)
// will send as a disappearing message
await sock.sendMessage(jid, { text: 'hello' }, { ephemeralExpiration: WA_DEFAULT_EPHEMERAL })
// turn off disappearing messages
await sock.sendMessage(
    jid,
    { disappearingMessagesInChat: false }
)
```

Misc

- To check if a given ID is on WhatsApp

```
 const id = '123456'
 const [result] = await sock.onWhatsApp(id)
 if (result.exists) console.log (`${id} exists on WhatsApp, as jid: ${result.jid}`)
```

- To query chat history on a group or with someone TODO, if possible
- To get the status of some person

```
 const status = await sock.fetchStatus("xyz@s.whatsapp.net")
 console.log("status: " + status)
```

- To change your profile status

```
 const status = 'Hello World!'
 await sock.updateProfileStatus(status)
```

- To change your profile name

```
 const name = 'My name'
 await sock.updateProfileName(name)
```

- To get the display picture of some person/group

```
 // for low res picture
 const ppUrl = await sock.profilePictureUrl("xyz@g.us")
 console.log("download profile picture from: " + ppUrl)
 // for high res picture
 const ppUrl = await sock.profilePictureUrl("xyz@g.us", 'image')
```

- To change your display picture or a group's

```
const jid = '111234567890-1594482450@g.us' // can be your own too
await sock.updateProfilePicture(jid, { url: './new-profile-picture.jpeg' })
```

- To remove your display picture or a group's

```
const jid = '111234567890-1594482450@g.us' // can be your own too
await sock.removeProfilePicture(jid)
```

- To get someone's presence (if they're typing or online)

```
// the presence update is fetched and called here
sock.ev.on('presence.update', json => console.log(json))
// request updates for a chat
await sock.presenceSubscribe("xyz@s.whatsapp.net")
```

- To block or unblock user

```
await sock.updateBlockStatus("xyz@s.whatsapp.net", "block") // Block user
await sock.updateBlockStatus("xyz@s.whatsapp.net", "unblock") // Unblock user
```

- To get a business profile, such as description or category

```
const profile = await sock.getBusinessProfile("xyz@s.whatsapp.net")
console.log("business description: " + profile.description + ", category: " + profile.category)
```

  Of course, replace `xyz` with an actual ID.

Groups

- To create a group

```
// title & participants
const group = await sock.groupCreate("My Fab Group", ["1234@s.whatsapp.net", "4564@s.whatsapp.net"])
console.log ("created group with id: " + group.gid)
sock.sendMessage(group.id, { text: 'hello there' }) // say hello to everyone on the group
```

- To add/remove people to a group or demote/promote people

```
// id & people to add to the group (will throw error if it fails)
const response = await sock.groupParticipantsUpdate(
    "abcd-xyz@g.us",
    ["abcd@s.whatsapp.net", "efgh@s.whatsapp.net"],
    "add" // replace this parameter with "remove", "demote" or "promote"
)
```

- To change the group's subject

```
await sock.groupUpdateSubject("abcd-xyz@g.us", "New Subject!")
```

- To change the group's description

```
await sock.groupUpdateDescription("abcd-xyz@g.us", "New Description!")
```

- To change group settings

```
// only allow admins to send messages
await sock.groupSettingUpdate("abcd-xyz@g.us", 'announcement')
// allow everyone to send messages
await sock.groupSettingUpdate("abcd-xyz@g.us", 'not_announcement')
// allow everyone to modify the group's settings -- like display picture etc.
await sock.groupSettingUpdate("abcd-xyz@g.us", 'unlocked')
// only allow admins to modify the group's settings
await sock.groupSettingUpdate("abcd-xyz@g.us", 'locked')
```

- To leave a group

```
await sock.groupLeave("abcd-xyz@g.us") // (will throw error if it fails)
```

- To get the invite code for a group

```
const code = await sock.groupInviteCode("abcd-xyz@g.us")
console.log("group code: " + code)
```

- To revoke the invite code in a group

```
const code = await sock.groupRevokeInvite("abcd-xyz@g.us")
console.log("New group code: " + code)
```

- To query the metadata of a group

```
const metadata = await sock.groupMetadata("abcd-xyz@g.us")
console.log(metadata.id + ", title: " + metadata.subject + ", description: " + metadata.desc)
```

- To join the group using the invitation code

```
const response = await sock.groupAcceptInvite("xxx")
console.log("joined to: " + response)
```

  Of course, replace `xxx` with invitation code.

- To get group info by invite code

```
const response = await sock.groupGetInviteInfo("xxx")
console.log("group information: " + response)
```

- To join the group using groupInviteMessage

```
const response = await sock.groupAcceptInviteV4("abcd@s.whatsapp.net", groupInviteMessage)
console.log("joined to: " + response)
```

  Of course, replace `xxx` with invitation code.

- To get list request join

```
const response = await sock.groupRequestParticipantsList("abcd-xyz@g.us")
console.log(response)
```

- To approve/reject request join

```
const response = await sock.groupRequestParticipantsUpdate(
    "abcd-xyz@g.us", // id group,
    ["abcd@s.whatsapp.net", "efgh@s.whatsapp.net"],
    "approve" // replace this parameter with "reject"
)
console.log(response)
```

Privacy

- To get the privacy settings

```
const privacySettings = await sock.fetchPrivacySettings(true)
console.log("privacy settings: " + privacySettings)
```

- To update the LastSeen privacy

```
const value = 'all' // 'contacts' | 'contact_blacklist' | 'none'
await sock.updateLastSeenPrivacy(value)
```

- To update the Online privacy

```
const value = 'all' // 'match_last_seen'
await sock.updateOnlinePrivacy(value)
```

- To update the Profile Picture privacy

```
const value = 'all' // 'contacts' | 'contact_blacklist' | 'none'
await sock.updateProfilePicturePrivacy(value)
```

- To update the Status privacy

```
const value = 'all' // 'contacts' | 'contact_blacklist' | 'none'
await sock.updateStatusPrivacy(value)
```

- To update the Read Receipts privacy

```
const value = 'all' // 'none'
await sock.updateReadReceiptsPrivacy(value)
```

- To update the Groups Add privacy

```
const value = 'all' // 'contacts' | 'contact_blacklist'
await sock.updateGroupsAddPrivacy(value)
```

- To update the Default Disappearing Mode

```
const duration = 86400 // 604800 | 7776000 | 0
await sock.updateDefaultDisappearingMode(duration)
```

Broadcast Lists & Stories

Messages can be sent to broadcasts & stories. you need to add the following message options in sendMessage, like this:

```
sock.sendMessage(jid, {image: {url: url}, caption: caption}, {backgroundColor : backgroundColor, font : font, statusJidList: status
```

- the message body can be a extendedTextMessage or imageMessage or videoMessage or voiceMessage
- You can add backgroundColor and other options in the message options
- broadcast: true enables broadcast mode
- statusJidList: a list of people that you can get which you need to provide, which are the people who will get this status message.
- You can send messages to broadcast lists the same way you send messages to groups & individual chats.
- Right now, WA Web does not support creating broadcast lists, but you can still delete them.
- Broadcast IDs are in the format `12345678@broadcast`
- To query a broadcast list's recipients & name:

```
const bList = await sock.getBroadcastListInfo("1234@broadcast")
console.log (`list name: ${bList.name}, recps: ${bList.recipients}`)
```

Writing Custom Functionality

Baileys is written with custom functionality in mind. Instead of forking the project & re-writing the internals, you can simply write your own extensions.

First, enable the logging of unhandled messages from WhatsApp by setting:

```
const sock = makeWASocket({
    logger: P({ level: 'debug' }),
})
```

This will enable you to see all sorts of messages WhatsApp sends in the console.

Some examples:

1. Functionality to track the battery percentage of your phone. You enable logging and you'll see a message about your battery pop up in the console: `{"level":10,"fromMe":false,"frame":{"tag":"ib","attrs":{"from":"@s.whatsapp.net"},"content":[{"tag":"edge_routing","attrs":{},"content":[{"tag":"routing_info","attrs":{},"content":{"type":"Buffer","data":[8,2,8,5]}}]}]},"msg":"communication"}`

   The "frame" is what the message received is, it has three components:

   - `tag` -- what this frame is about (eg. message will have "message")
   - `attrs` -- a string key-value pair with some metadata (contains ID of the message usually)
   - `content` -- the actual data (eg. a message node will have the actual message content in it)
   - read more about this format here

   You can register a callback for an event using the following:

```
// for any message with tag 'edge_routing'
sock.ws.on(`CB:edge_routing`, (node: BinaryNode) => { })
// for any message with tag 'edge_routing' and id attribute = abcd
sock.ws.on(`CB:edge_routing,id:abcd`, (node: BinaryNode) => { })
// for any message with tag 'edge_routing', id attribute = abcd & first content node routing_info
sock.ws.on(`CB:edge_routing,id:abcd,routing_info`, (node: BinaryNode) => { })
```

Also, this repo is now licenced under GPL 3 since it uses libsignal-node