Go Proton API

Official Go client library for the Proton REST API — covering Mail, Contacts, Calendar, Drive Shares, and account operations.

- Overview & Quick Start
  - Overview & Quick Start
- Authentication & Sessions
  - Authentication & Sessions
- Mail & Messages
  - Mail & Messages
- Contacts, Calendar & Labels
  - Contacts
  - Calendar
  - Labels & Addresses
- Attachments, Shares & Events
  - Attachments & Drive Shares
  - Real-time Events & User Account

Overview & Quick Start

Overview & Quick Start

Go Proton API

Overview

`github.com/ProtonMail/go-proton-api` is the official Go library implementing a client for (a subset of) the Proton REST API. It covers **Mail, Contacts, Calendar, Drive Shares**, and core account operations. Built on `resty/v2` for HTTP communication and `gopenpgp/v2` for PGP encryption/decryption.

Quick Start

```
import (
    "context"
    "github.com/ProtonMail/go-proton-api"
)

// 1. Create a Manager (shared across clients)
m := proton.New()
defer m.Close()

ctx := context.Background()

// 2a. Login with username/password (full SRP flow)
c, auth, err := m.NewClientWithLogin(ctx, "user@proton.me", []byte("password"))
if err != nil {
    panic(err)
}
defer c.Close()
```

```
// Handle 2FA if needed
if auth.TwoFA.Enabled&proton.HasTOTP != 0 {
    if err := c.Auth2FA(ctx, proton.Auth2FAReq{TwoFactorCode: "123456"}); err != nil {
        panic(err)
    }
}

// 2b. Or create from existing tokens (no login needed)
c := m.NewClient("uid", "accessToken", "refreshToken")
defer c.Close()
```

Installation

```
go get github.com/ProtonMail/go-proton-api
```

Requires **Go 1.26+**.

Architecture

The library uses a **Manager → Client** hierarchy:

- **Manager** — Top-level factory object. Created via `New(opts ...Option)`. Holds a shared `*resty.Client`, status observers, error handlers, and panic recovery. Shared across all clients.
- **Client** — Per-user client bound to a specific UID. Created by the Manager via `NewClient()`, `NewClientWithLogin()`, or `NewClientWithRefresh()`. Each Client handles one Proton account.

Core Features

Feature	Description
Automatic auth refresh	401 responses trigger transparent token refresh + retry
Connection status	`Manager.AddStatusObserver()` receives `StatusUp` / `StatusDown` notifications
Error handlers	`Manager.AddErrorHandler(code, handler)` — register callbacks for specific API error codes
Request hooks	Per-client and global pre/post request middleware via resty

Feature	Description
Parallel operations	Delete messages, import messages, and attachment downloads all use parallel execution
Undo support	Label/unlabel operations return undo tokens for reversible actions
Paging helpers	`GetAllContacts()`, `GetAllMessageIDs()` etc. auto-page through results

Authentication & Sessions

Authentication & Sessions

Authentication & Sessions

The library handles the full authentication lifecycle: SRP-based login, 2FA/FIDO2, token refresh, session management, and user creation.

Login Methods

SRP Login (Username/Password)

Full Secure Remote Password login flow -- zero-knowledge password authentication:

```
m := proton.New()
ctx := context.Background()

c, auth, err := m.NewClientWithLogin(ctx, "user@proton.me", []byte("password"))
if err != nil {
    panic(err)
}
defer c.Close()
```

2FA Support

If the account has 2FA enabled:

```
if auth.TwoFA.Enabled&proton.HasTOTP != 0 {
    if err := c.Auth2FA(ctx, proton.Auth2FAReq{TwoFactorCode: "123456"}); err != nil {
        panic(err)
    }
}
```

FIDO2 Support

```
if auth.TwoFA.Enabled&proton.HasFIDO2 != 0 {
    if err := c.Auth2FA(ctx, proton.Auth2FAReq{
        FIDO2Data: proton.FIDO2Req{
            Attestation: "...",
            AuthenticatorData: "...",
            Signature: "...",
            CredentialID: "...",
        },
    }); err != nil {
        panic(err)
    }
}
```

Refresh Token Login

Create a client from existing refresh token (no password needed):

```
c, _, err := m.NewClientWithRefresh(ctx, "uid", "refreshToken")
if err != nil {
    panic(err)
}
defer c.Close()
```

Direct Token Login

Create a client when all auth info is already known:

```
c := m.NewClient("uid", "accessToken", "refreshToken")
defer c.Close()
```

Session Management

```
// List active sessions
sessions, err := c.AuthSessions(ctx)

// Revoke a specific session
err := c.AuthRevoke(ctx, authUID)

// Revoke all sessions
err := c.AuthRevokeAll(ctx)

// Delete current session
err := c.AuthDelete(ctx)
```

Auth Handlers

Register callbacks to handle auth events:

```
// Save new tokens when they are refreshed
c.AddAuthHandler(func(auth proton.Auth) {
    // Save auth.AccessToken, auth.RefreshToken to keychain
})

// Handle de-authentication
c.AddDeauthHandler(func() {
    // Clear stored credentials
})
```

Key Types

Type	Description
Auth	Contains UID, AccessToken, RefreshToken, ServerProof, Scope, TwoFA info
AuthInfo	Server-provided auth metadata: salt, modulus, server ephemeral, 2FA status
TwoFAInfo / TwoFAStatus	Enum for TOTP, FIDO2, or both
FIDO2Req	FIDO2 authentication data (attestation, authenticator data, signature, credential ID)

Type	Description
AuthSession	Active session with client info and revocability flag
PasswordMode	One-password vs two-password mode

Security Notes

- Login uses SRP (Secure Remote Password) protocol via go-srp for zero-knowledge password authentication
- By default, the Manager verifies the server expected proof (verifyProofs flag) to detect MITM attacks
- The server sends salt/modulus/ephemeral; the client computes proofs and verifies the server proof

Mail & Messages

Mail & Messages

Mail & Messages

Complete mail operations: fetching messages (metadata + full bodies), labeling, marking read/unread/forwarded, deleting, creating/updating/sending drafts, importing messages, building RFC822 MIME messages, and encrypting/decrypting with PGP.

Fetching Messages

Single Message

```
msg, err := c.GetMessage(ctx, messageID)
// msg contains: ID, addressID, labelIDs, subject, sender/recipients, flags, timestamps
```

Full Message with Attachments

```
fullMsg, err := c.GetFullMessage(ctx, messageID, scheduler, storageProvider)
// fullMsg.Message + fullMsg.AttData ([][]byte of decoded attachment data)
```

Message Metadata (List)

```
// Get all message metadata
meta, err := c.GetMessageMetadata(ctx, proton.MessageFilter{
    LabelIDs: []string{"inbox-label-id"},
})

// Paginated with filters
meta, err := c.GetMessageMetadataPage(ctx, page, pageSize, filter)
```

```
// Count messages
count, err := c.CountMessages(ctx)

// Get all message IDs (auto-paginating)
allIDs, err := c.GetAllMessageIDs(ctx, afterID)
```

Drafts

```
// Create a draft
draft, err := c.CreateDraft(ctx, addrKR, proton.CreateDraftReq{
    Subject: "Hello",
    Sender:  proton.Address{ID: "address-id"},
    To:      []proton.Address{{Email: "recipient@example.com"}},
    Body:    "Message body",
    MIMEType: "text/plain",
})

// Update a draft
updated, err := c.UpdateDraft(ctx, draftID, addrKR, proton.UpdateDraftReq{
    Body: "Updated body",
    Action: proton.DraftActionReply, // reply/reply-all/forward/auto-response/read-receipt
})

// Send a draft
err := c.SendDraft(ctx, draftID, proton.SendDraftReq{
    // MessagePackage with encrypted body + per-recipient key packets
})
```

Bulk Operations

```
// Delete messages (parallel chunking, max 1000 per page)
err := c.DeleteMessage(ctx, id1, id2, id3...)

// Mark as read/unread
```

```
err := c.MarkMessagesRead(ctx, ids...)
err := c.MarkMessagesUnread(ctx, ids...)

// Mark as forwarded/unforwarded
err := c.MarkMessagesForwarded(ctx, ids...)
err := c.MarkMessagesUnForwarded(ctx, ids...)

// Label/unlabel with undo support
err := c.LabelMessages(ctx, ids, labelID)
err := c.UnlabelMessages(ctx, ids, labelID)
// Returns undo tokens for reversible actions
```

Message Import

Bulk import of raw RFC822 messages:

```
stream := c.ImportMessages(ctx, addrKR, workers, buffer, req1, req2, ...)
// req := proton.ImportReq{
//     RawMessage: []byte("Raw RFC822 message..."),
//     AddressID:  "address-id",
//     LabelIDs:   []string{"label-id"},
// }

for res := range stream {
    fmt.Println(res.ID, res.Error)
}
```

MIME Building & Encryption

```
// Reconstruct full RFC822 MIME from decrypted parts
rawMIME, err := proton.BuildRFC822(kr, msg, attData)

// Encrypt a raw RFC822 message
encrypted, err := proton.EncryptRFC822(kr, literal)
```

Key Types

Type	Description
`MessageMetadata`	ID, addressID, labelIDs, subject, sender/recipients, flags, timestamps
`Message`	Extends Metadata with raw Header, Headers, encrypted Body, MIMEType, Attachments
`FullMessage`	Message + decoded attachment data
`MessageFilter`	Filter by ID list, subject, addressID, externalID, labelID
`MessageFlag`	Bitmask: received, sent, internal, E2E, replied, forwarded, spam/DMARC/DKIM/SPF
`EncryptionScheme`	InternalScheme, EncryptedOutsideScheme, ClearScheme, PGPInlineScheme, PGPMIMEScheme
`SendPreferences`	Per-recipient encryption/signature/MIME type preferences
`MessagePackage`	Encrypted message body + per-recipient key packets
`DraftTemplate`	Subject, sender, recipients, body, MIME type for draft creation
`ImportReq` / `ImportMetadata`	Import request with raw message bytes and metadata

Contacts, Calendar & Labels

Contacts

Contacts

CRUD operations for contacts and contact emails via `/contacts/v4`.

Operations

Get Contacts

```
// Single contact by ID
contact, err := c.GetContact(ctx, contactID)

// Count contacts
count, err := c.CountContacts(ctx)

// Paginated listing
contacts, err := c.GetContacts(ctx, page, pageSize)

// Auto-paginating - get all
allContacts, err := c.GetAllContacts(ctx)

// Auto-paginating with custom page size
allContacts, err := c.GetAllContactsPaged(ctx, pageSize)
```

Search by Email

```
// Count matches
count, err := c.CountContactEmails(ctx, "user@example.com")
```

```
// Paginated search
emails, err := c.GetContactEmails(ctx, "user@example.com", page, pageSize)

// Get all matching
allEmails, err := c.GetAllContactEmails(ctx, "user@example.com")
```

Create / Update / Delete

```
// Bulk create with overwrite and label options
results, err := c.CreateContacts(ctx, proton.CreateContactsReq{
    Contacts: []proton.Contact{
        {
            Cards: []proton.ContactCard{...},
            Email: "user@example.com",
        },
    },
})

// Update a contact
updated, err := c.UpdateContact(ctx, contactID, proton.UpdateContactReq{
    Cards: []proton.ContactCard{...},
})

// Bulk delete
err := c.DeleteContacts(ctx, proton.DeleteContactsReq{
    IDs: []string{id1, id2},
})
```

Per-Contact Encryption Preferences

Contact settings are stored as VCard-embedded X-PM-* fields:

```
// Get encryption settings for a contact
settings := contact.GetSettings()
```

```
// Returns: MIME type, PGP scheme (inline/MIME), sign/encrypt flags, PGP keys

// Set encryption settings
contact.SetSettings(proton.ContactSettings{
    MIMEType:       "text/plain",
    Scheme:         proton.PGPMIMEScheme,
    Sign:           true,
    Encrypt:        true,
    EncryptUntrusted: true,
    PGPKeys:        []proton.PGPKey{...},
})
```

Key Types

Type	Description
`Contact`	Composite of ContactMetadata and ContactCards
`ContactMetadata`	ID, name, UID, size, timestamps, emails, labels
`ContactCards`	VCard data for the contact
`ContactEmail`	Email entry with ID, name, email string, type tags, contactID, labels
`ContactSettings`	Per-contact encryption preferences (MIME type, scheme, sign/encrypt flags, PGP keys)
`RecipientType`	Internal vs external recipient classification

Calendar

Calendar

Calendar and calendar event retrieval via `/calendar/v1`. Includes key management for encrypted calendar data.

Operations

Calendars

```
// List all calendars
calendars, err := c.GetCalendars(ctx)

// Get a single calendar
cal, err := c.GetCalendar(ctx, calendarID)

// Get encryption keys for a calendar
keys, err := c.GetCalendarKeys(ctx, calendarID)

// Get shared calendar members
members, err := c.GetCalendarMembers(ctx, calendarID)

// Get passphrase for calendar key decryption
passphrase, err := c.GetCalendarPassphrase(ctx, calendarID)
```

Calendar Events

```
// Count events
count, err := c.CountCalendarEvents(ctx, calendarID)
```

```
// Paginated listing with optional filters
events, err := c.GetCalendarEvents(ctx, calendarID, page, pageSize, filter)

// Auto-paginating - get all events
allEvents, err := c.GetAllCalendarEvents(ctx, calendarID, filter)

// Single event by ID
event, err := c.GetCalendarEvent(ctx, calendarID, eventID)
```

Decryption

Calendar events use **two-layer encryption**:

1. Calendar keys (passphrase-locked)
2. Shared events add a second layer of sharing key packets

```
// Unlock calendar keys with passphrase
keyRing, err := keys.Unlock(passphrase)

// Decrypt individual event parts
decoded, err := eventPart.Decode(calKR, addrKR, keyPacket)
// Handles both two-layer decryption and PGP signature verification
```

Key Types

Type	Description
`Calendar`	ID, name, description, color, display flag, type (normal/subscribed), flags
`CalendarKey`	Encrypted private key with `Unlock(passphrase)` returning `*crypto.Key`
`CalendarKeys`	Slice of keys with `Unlock(passphrase)` returning `*crypto.KeyRing`
`CalendarMember`	ID, permissions, email, color, display, calendarID
`CalendarPassphrase`	Encrypted passphrase with `Decrypt(memberID, addrKR)` for decryption

Type	Description
`CalendarEvent`	UID, start/end times, timezone, full-day flag, author, attendees, encrypted parts
`CalendarEventPart`	Individual part with type (clear/encrypted/signed), data, signature, author
`EventAction`	Delete, Create, Update, UpdateFlags

PGP Signature Verification

Encrypted calendar parts can also be signed. The `Decode` method verifies signatures using the address keyring.

Labels & Addresses

Labels

Label (category/folder) management via `/core/v4/labels`.

Operations

```
// Fetch labels filtered by type(s)
labels, err := c.GetLabels(ctx, proton.LabelLabel, proton.LabelFolder)

// Find a specific label by ID
label, err := c.GetLabel(ctx, labelID, proton.LabelLabel)

// Create a label
label, err := c.CreateLabel(ctx, proton.CreateLabelReq{
    Name:   "Personal",
    Color:  "#ff0000",
    Type:   proton.LabelLabel,
    Parent: parentID,
})

// Update a label
err := c.UpdateLabel(ctx, labelID, proton.UpdateLabelReq{
    Name: "Updated Name",
    Color: "#00ff00",
})

// Delete a label
err := c.DeleteLabel(ctx, labelID)
```

System Label Constants

Constant	ID	Description
`InboxLabel`	"0"	Inbox
`AllSentLabel`	"2"	All Sent
`TrashLabel`	"3"	Trash
`SpamLabel`	"4"	Spam
`AllMailLabel`	"5"	All Mail
`StarredLabel`	"10"	Starred

Email Addresses

Email address management via `/core/v4/addresses`.

Operations

```
// List addresses (sorted by order)
addresses, err := c.GetAddresses(ctx)

// Get a single address
addr, err := c.GetAddress(ctx, addressID)

// Reorder addresses
err := c.OrderAddresses(ctx, proton.OrderAddressesReq{...})

// Enable/disable address
err := c.EnableAddress(ctx, addressID)
err := c.DisableAddress(ctx, addressID)

// Delete address
err := c.DeleteAddress(ctx, addressID)
```

Address Types

Type	Description
`Original`	Original Proton address

Type	Description
`Alias`	Proton alias
`Custom`	Custom domain address
`Premium`	Premium address
`External`	Bring Your Own Email (BYOE)

BYOE Detection

```
if addr.IsBYOEAddress() {
    // External type with sending enabled
}
```

Key Types

Type	Description
`Label`	ID, parentID, name, path ([]string), color, type
`LabelType`	Label, ContactGroup, Folder, System
`Address`	ID, email, send/receive flags, status, type, order, display name, keys
`AddressStatus`	Disabled, enabled, deleting
`AddressType`	Original, alias, custom, premium, external

Path Handling

The API sends label path as a string (e.g., `"Inbox/SubFolder"`), which is split into a `[]string` slice for Go use, with custom `MarshalJSON`/`UnmarshalJSON`.

Attachments, Shares & Events

Attachments & Drive Shares

Attachments

Attachment upload and download via `/mail/v4/attachments`.

Upload

```
attachment, err := c.UploadAttachment(ctx, addrKR, proton.CreateAttachmentReq{
    MessageID:  "message-id",
    Filename:   "document.pdf",
    MIMEType:   "application/pdf",
    Disposition: proton.AttachmentDisposition,
    Body:       fileBytes,
})
```

Upload is **encrypted with the address keyring** and **signed (detached)** before upload. The server stores the encrypted `KeyPacket`, `DataPacket`, and `Signature` separately.

Download

```
// Download as bytes
data, err := c.GetAttachment(ctx, attachmentID)

// Stream into io.ReaderFrom
err := c.GetAttachmentInto(ctx, attachmentID, reader)
```

Key Types

Type	Description
`Attachment`	ID, name, size, MIME type, disposition (inline/attachment), headers, key packets, signature
`CreateAttachmentReq`	MessageID, filename, MIME type, disposition, content ID, body bytes
`Disposition`	InlineDisposition, AttachmentDisposition

Drive Shares

Proton Drive share management via `/drive/shares`.

Operations

```
// List shares
shares, err := c.ListShares(ctx, all)

// Get a single share with full key material
share, err := c.GetShare(ctx, shareID)
```

Key Derivation

Shares use a two-step PGP decryption + passphrase unlock process:

```
// Decrypt passphrase with address keyring, verify signature, unlock share key
keyRing, err := share.GetKeyRing(addrKR)
// Returns *crypto.KeyRing for Drive access
```

Key Types

Type	Description
`ShareMetadata`	ShareID, linkID, volumeID, type, state, creation/modify times, creator email

Type	Description
`Share`	Extends metadata with addressID, addressKeyID, encrypted key, passphrase, passphrase signature
`ShareType`	Main (1), Standard (2), Device (3)
`ShareState`	Active (1), Deleted (2)

Real-time Events & User Account

Real-time Events

Event polling and streaming for real-time notifications via `/core/v4/events`.

Event Streaming

```
// Get the latest event ID
fromEventID, err := c.GetLatestEventID(context.Background())

// Create a new event streamer
for event := range c.NewEventStream(ctx, 20*time.Second, 20*time.Second, fromEventID) {
    fmt.Println(event.EventID)
    // Process event.User, event.Messages, event.Labels, etc.
}
```

The event stream uses a custom `NewTicker` with random jitter to avoid thundering herd.

Event Polling

```
// Fetch events starting after eventID
events, more, err := c.GetEvent(ctx, eventID)
// Returns up to 50 events per call with more indicating continuation
```

Event Types

The `Event` struct contains typed sub-events:

```
event.User          // User changes
event.UserSettings  // User settings changes
event.MailSettings  // Mail settings changes
event.Messages      // Message changes ([]MessageEvent)
event.Labels        // Label changes ([]LabelEvent)
event.Addresses     // Address changes ([]AddressEvent)
event.Notifications // Notifications ([]NotificationEvent)
event.UsedSpace     // Storage usage changes
```

Key Types

Type	Description
`Event`	Contains eventID, refresh flags, and typed sub-events
`RefreshFlag`	RefreshMail (1), RefreshAll (255)
`EventAction`	Delete, Create, Update, UpdateFlags
`MessageEvent`	EventItem (ID + Action) with full Message entity
`LabelEvent`	EventItem (ID + Action) with full Label entity
`AddressEvent`	EventItem (ID + Action) with full Address entity

User Account

User account operations via `/core/v4/users`.

Operations

```
// Fetch current user info
user, err := c.GetUser(ctx)

// With human verification token
user, err := c.GetUserWithHV(ctx, hvToken)
```

```
// Delete account (requires SRP proof re-authentication)
err := c.DeleteUser(ctx, password, proton.DeleteUserReq{
    Reason:  "no longer needed",
    Feedback: "test account",
})
```

Key Types

Type	Description
`User`	ID, name, display name, email, keys, used/max/upload space limits, credit, currency
`ProductUsedSpace`	Space breakdown by calendar, contact, drive, mail, pass (password manager)
`DeleteUserReq`	Reason, feedback, email confirmation

Feature Flags

Fetch feature flag/toggle results from the Proton API:

```
features, err := c.GetFeatures(ctx, "sticky-key-uuid")
// Returns FeatureFlagResult for a given UUID sticky key
```