TigaseDoc

Release 0.1

Tigase, Inc.

Jun 21, 2023

CONTENTS

CHAPTER

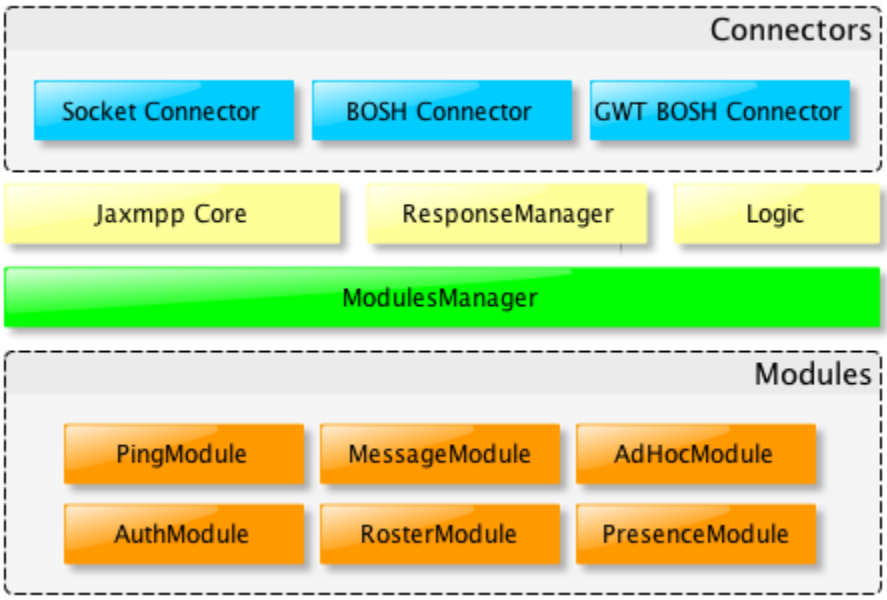
ONE

DESIGN AND IMPLEMENTATION

1.1 Tigase Halcyon

Halcyon is a multiplatform extensible XMPP client library

1.2 Design



CHAPTER

TWO

QUICKSTART

2.1 Simplest client

Here is example of simplest client sending one message.

SimplestClient.kt.

```
val halcyon = Halcyon()
halcyon.configuration.let {
    it.setJID("client@tigase.net".toBareJID())
    it.setPassword("secret")
}
halcyon.connectAndWait()

halcyon.request.message {
    to = "romeo@example.net".toJID()
    "body"{
        +"Art thou not Romeo, and a Montague?"
    }
}.send()

halcyon.disconnect()
```

2.2 Handling changes of connection status

We can listen for changing status of connection:

```
halcyon.eventBus.register<HalcyonStateChangeEvent>(HalcyonStateChangeEvent.TYPE) {␣
↪stateChangeEvent ->
    println("Halcyon state: ${stateChangeEvent.oldState}->${stateChangeEvent.newState}")
}
```

Available states:

- `Connecting` - this state means, that method `connect()` was called, and connection to server is in progress.
- `Connected` - connection is fully established.
- `Disconnecting` - connection is closing because of error or manual disconnecting.
- `Disconnected` - Halcyon is disconnected from XMPP server, but it is still active. It may start reconnecting to server automatically.

- `Stopped` - Halcyon is turned off (not active).

CHAPTER

THREE

WORKING WITH REQUESTS

Each module may perform some requests on other XMPP entities, and (if yes) must return `RequestBuilder` object to allow check status of request and receive response.

For example, suppose we want to ping XMPP server (as described in XEP-0199):

Sample ping request and response.

```
<!-- Client sends: -->
<iq to='tigase.net' id='ping-1' type='get'>
  <ping xmlns='urn:xmpp:ping'/>
</iq>

<!-- Client receives: -->
<iq from='tigase.net' to='client@tigase.net' id='ping-1' type='result'/>
```

There is module `PingModule` in Halcyon to do it:

Kotlin sample.

```
val pingModule: PingModule = client.modules[PingModule.TYPE]
val request = pingModule.ping("tigase.net".toJID()).send()
```

In this case, method `ping()` returns `RequestBuilder` to allow add result handler, change default timeout and other operations. To send stanza you have to call method 'send()'. There is also available method `build()` what also creates request object, but doesn't sends it.

Note: On JVM, methods of handler will be called from separate thread.

Most universal way to receive result in asynchronous way is add response handler to request builder:

```
val client = Halcyon()
val pingModule: PingModule = client.modules[PingModule.TYPE]
pingModule.ping("tigase.net".toJID()).response { result ->
    result.onSuccess { pong ->
        println("Pong: ${pong.time}ms")
    }
    result.onFailure { error ->
        println("Error $error")
    }
}.send()
```

CHAPTER

FOUR

WORKING WITH EVENTS

Halcyon is events driven library. It means you have to listen for events to receive message, react for disconnection or so. There is single events bus inside, to which you can register listeners. Each part of library (like modules, connectors, etc.) may have set of own events to fire.

General code to registering events:

```
halcyon.eventBus.register<EVENT_TYPE>(EVENT_NAME) { event ->
...
}
```

In Halcyon, name of event is defined as constant variable named `TYPE` in each event.

For example:

```
halcyon.eventBus.register<ReceivedXMLElementEvent>(ReceivedXMLElementEvent.TYPE) { event␣
↪->
    println(" >>> ${event.element.getAsString()}")
}
```

You can use EventBus for you own applications. No need to register events types. Just create object inherited from `tigase.halcyon.core.eventbus.Event` and call method `eventbus.fire()`:

```
data class SampleEvent(val sampleData: String) : Event(TYPE){

    companion object {
        const val TYPE = "sampleEvent"
    }

}

halcyon.eventBus.fire(SampleEvent("test"))
```

CHAPTER

FIVE

MODULES

Halcyon contains set of modules responsible for implementation of various tasks: sending and receiving messages, authentication, pinging, uploading files, handling roster, presences, etc.

Modules:

5.1 BindModule

Resource Binding module. This module is responsible for resource binding as described in RFC.

5.1.1 Properties

- `boundJID` - contains full JID bound during resource binding process, or null if client isn't logged in and/or not binded.

5.1.2 Methods

There is no reason to call methods from this module in client. This module is used internally by Halcyon library.

`bind(String)`

Method prepares request to bind resource. As parameters it gets proposed resource name or `null` if resource name should be generated by server. In response it returns object `BindResult` contains full bound JID.

5.2 DiscoveryModule

This module implements XEP-0030: Service Discovery.

5.2.1 Properties

There are few properties to set in this module:

- `clientName` - Client name.
- `clientVersion` - Version of client.
- `clientCategory` - Category of client.
- `clientType` - Type of client.

Category and type of client are described in Service Discovery Identities document.

5.2.2 Events

`ServerFeaturesReceivedEvent`

Fired when server features, to where client is connected, are received. Client asks for server features automatically during login process.

`AccountFeaturesReceivedEvent`

Fired when user account features are received. Client asks for those features automatically.

5.2.3 Methods

`info(JID, String)`

Method prepares `disco#info` request. As parameters it takes JID of entity and node name. Both are optional. In response returns object `Info` contains JID of entity, node name, list of identities and list of features.

`items(JID, String)`

Method prepares `disco#items` request. As parameters it takes JID of entity and node name. Both are optional. In response returns object `Items` contains JID of entity, node name and list of items.

`findComponent((Info) → Boolean, (Info) → Unit)`

This method may be used to find component on currently connected server with specific features or type.

As first parameter it takes condition checker, which much check if given `Info` is this object what we are searching for. Second parameter is consumer.

```
findComponent({ candidate ->
                candidate.identities.any { it.type == "mix" }
            }) { result ->
    println("${result.jid}")
}
```

5.3 MessageCarbonsModule

This module implements XEP-0280: Message Carbons.

5.3.1 Events

Sent

Fired when client receives carbon of message sent by other entity using the same account. Event contains carboned message.

Received

Fired when client receives carbon of message received by other entity using the same account. Event contains carboned message.

5.3.2 Methods

enable()

Method prepares request to enable carbon messages in current session.

disable()

Method prepares request to disable carbon messages in current session.

5.4 PingModule

This module implements XEP-0199: XMPP Ping. It allows to ping XMPP entities over XML stream.

5.4.1 Published features

- `urn:xmpp:ping`

5.4.2 Methods

ping(JID)

This method prepares ping request. Response object `Pong` contains measured round-trip time in miliseconds.

5.5 PresenceModule

Module for handling received presence information.

5.5.1 Events

PresenceReceivedEvent

Fired when any Presence stanza is received by client.

Fields:

- `jid` - Stanza sender JID.
- `stanzaType` - Presence stanza type.
- `stanza` - Whole received presence stanza.

ContactChangeStatusEvent

Fired when Presence stanza is received but it contains different set of fields:

- `jid` - Bare JID of contact.
- `status` - Human readable status set by contact.
- `presence` - Current "best" presence stanza, based on presence priority.
- `lastReceivedPresence` - Just received presence stanza.

Note that `presence` in this event may contain stanza received long time ago. Current event is caused by receiving presence from entity with lower priority.

5.5.2 Methods

getPresenceOf(jid: JID)

Returns presence of given entity or null if never received presence from this entity.

getBestPresenceOf(jid: BareJID)

Returns best known presence of given bare JID.

sendPresence(jid: JID?, type: PresenceType?, show: Show?, status: String)

Sends presence stanza to specific jid.

`sendSubscriptionSet(jid: JID, presenceType: PresenceType)`

Method for quick send response for subscription request.

5.6 RosterModule

This module implements XEP-0060: Publish-Subscribe. It adds publish-subscribe functionality.

5.6.1 Events

PubSubEventReceivedEvent

5.6.2 Methods

create(pubSubJID: JID, node: String, configForm: JabberDataForm? = null)

fun subscribe(pubSubJID: JID, node: String, jid: JID)

fun purgeItems(pubSubJID: JID, node: String)

fun retrieveSubscriptions(pubSubJID: JID, node: String)

fun modifySubscriptions(pubSubJID: JID, node: String, subscriptions: List<Subscription>)

fun deleteItem(jid: JID, node: String, itemId: String)

fun retrieveItem(jid: JID, node: String, itemId: String? = null)

fun publish(jid: JID?, node: String, itemId: String?, payload: Element? = null)

fun retrieveAffiliations(jid: JID?, node: String? = null)

5.7 RosterModule

Module is reponsible for keeping and managing roster items.

5.7.1 Events

`ItemAdded`

Fired when new item is added to roster.

ItemUpdated

Fired when item is modified.

ItemRemoved

Fired when item is removed from roster.

5.7.2 Methods

addItem(vararg items: RosterItem)

Method prepares request which add or update item to roster. When server confirms action, event `ItemAdded` or `ItemUpdated` will be fired.

deleteItem(vararg jids: BareJID)

Method prepares request to remove item from roster. When server confirms operation, event `ItemRemoved` will be fired.

getAllItems()

Method returns add known roster items.

5.7.3 Implementing own storage

RosterModule supports roster versioning, but it requires custom implementation of `RosterStore` to allow storing roster locally. By default, Halcyon has in-memory roster store. To do that, there is interface `tigase.halcyon.core.xmpp.modules.roster.RosterStore` what need to be extend. To use custom implementation of `RosterStore` simply put it to property `store` in `RosterModule`. Note, that it have to be done before login.

5.8 SASLModule

Module is responsible for whole client authentication process.

5.8.1 Properties

- `saslContext` contains context of module. Context is cleared when connection is started. It has several fields to read:
  - `mechanism` - used SASL mechanism,
  - `state` - current state of authentication process,
  - `complete` - `true` if authentication process is finished (it doesn't matter with success or with error).

5.8.2 Events

SASLStarted

Fired when authentication process begins.

Fields:

- `mechanism` - name of used SASL mechanism.

SASLSuccess

Fired when authentication is successful.

SASLError

Fired when authentication finished with error.

Fields:

- `error` - enum with type of error. SASL errors are described if RFC.
- `description` - human readable description of error (if provided by server).

5.8.3 Methods

There is no reason to call methods from this module in client. This module is used internally by Halcyon library.

startAuth()

This method begins authentication process. It doesn't return Request object.

5.9 VCardModule

This module allows to publishing and retrieving VCard4 as described in XEP-0292.

5.9.1 Properties

- `autoRetrieve` - If `true` then module automatically retrieve VCard before firing `VCardUpdatedEvent`. It is `false` by default.

5.9.2 Events

VCardUpdatedEvent

Fired when VCard update is received from PEP. Contains JID that the update applies to. If `autoRetrieve` is set to `true` then event will contains current VCard.

5.9.3 Methods

retrieveVCard(jid: BareJID)

This method prepare request for retrieving VCard of given JID. As result returns `VCard` object.

fun publish(vcard: VCard)

This method prepare request for publishing own vcard.

5.9.4 Usage example

Retrieving VCard

```
val vCardModule = halcyon.getModule<VCardModule>(VCardModule.TYPE)!!
vCardModule.retrieveVCard("someone@server.im".toBareJID()).response { result ->
    result.onSuccess {vcard->
        println("""
            Received vcard:
            Name: ${vcard.formattedName}
            Name: ${vcard.structuredName?.given} ${vcard.structuredName?.surname}
            Nick: ${vcard.nickname}
            Birthday: ${vcard.birthday}
            TimeZone: ${vcard.timeZone}
        """.trimIndent())

        println()
        vcard.addresses.forEach {addr->
            println("""
                ${addr.street}
                ${addr.locality} ${addr.region} ${addr.code}
                ${addr.country}

            """.trimIndent())
        }

    }
    result.onFailure {
        println("Cannot retrieve VCard. Error: $it")
    }
}.send()
```

Publishing VCard

Publishing own vcard is very simple:

```
val vCardModule = halcyon.getModule<VCardModule>(VCardModule.TYPE)!!
vCardModule.publish(vcard).response { result ->
    result.onSuccess { println("VCard published") }
    result.onFailure { println("VCard NOT published") }
}.send()
```

VCard object is mutable and can be edited. To create new VCard instance you can use VCard builder:

```
val vCard = vcard {
    structuredName {
        given = "Alice"
        surname = "Carl"
    }
    nickname = "alice"
    email {
        parameters {
            pref = 1
            +"work"
        }
        text = "alice@organisation.com"
    }
}
```

5.10 BlockingCommandModule

This module implements XEP-0191: Blocking Command and XEP-0377: Spam Reporting

5.10.1 Events

Blocked

Event fired when new contact was blocked. It contains few properties:

- `jid` - blocked JabberID
- `reason` - reason of blocking.
- `text` - optional human-readable description of blocking reason.

Unblocked

Event fired, when contact was unblocked. It contains one property:

- `jid` - unblocked JabberID

UnblockedAll

Event fired when all contacts are unblocked.

```
halcyon.eventBus.register<BlockingCommandEvent>(BlockingCommandEvent.TYPE) { event ->
    when (event) {
        is BlockingCommandEvent.Blocked -> println("$event")
        is BlockingCommandEvent.Unblocked -> println("$event")
        is BlockingCommandEvent.UnblockedAll -> println("All blocked contacts are␣
↪unblocked now!")
    }
}
```

5.10.2 Methods

Samples:

retrieveList(JID, String)

Retrieves list of blocked contacts. It allows to get list of blocked list, without information about reason.

```
halcyon.getModule<BlockingCommandModule>(BlockingCommandModule.TYPE).retrieveList().
↪response {
    it.onSuccess {
        println("Blocked: $it")
    }
    it.onFailure { println("Oops!") }
}.send()
```

block

Blocks given contact.

Params:

- `jid` - BareJID of contact to block
- `reason` - reason of blocking (optional, default value is `NotSpecified`)
- `text` - optional, human-readable description of blocking

```
halcyon.getModule<BlockingCommandModule>(BlockingCommandModule.TYPE)
    .block("spammer@server.com".toBareJID(), Reason.Spam, "It is SPAMMER!!!")
    .response {
        it.onSuccess { println("Done") }
        it.onFailure { println("Oops!") }
    }.send()
```

unblock

Unblocks given contact.

Params:

- `jids` - BareJIDs of contacts to unblock

Note: If `jids` will be empty, all blocked contacts will be unblocked!

```
halcyon.getModule<BlockingCommandModule>(BlockingCommandModule.TYPE)
    .unblock("spammer@server.com".toBareJID())
    .response {
        it.onSuccess { println("Done") }
        it.onFailure { println("Oops!") }
    }.send()
```

unblockAll

Unblocks all blocked contacts.

```
halcyon.getModule<BlockingCommandModule>(BlockingCommandModule.TYPE)
    .unblockAll().response {
        it.onSuccess { println("Done") }
        it.onFailure { println("Oops!") }
    }.send()
```

5.11 CommandModule

This module implements XEP-0050: Ad-Hoc Commands.

5.11.1 Methods

retrieveCommandList

Retrieves list of commands allowed to execute on given XMPP entity. Because this command is just wrapper to `DiscoveryModule.items()` method, in response it returns `DiscoveryModule.Items` class.

retrieveCommandInfo

Retrieves detailed information about specified command. Because this command is just wrapper to `DiscoveryModule.info()` method, in response it returns `DiscoveryModule.Info` class.

executeCommand

This method executes Ad-hoc Command on XMPP entity specified by JabberID.

Arguments:

- `jid` - Jabber ID of command executor,
- `command` - command name,
- `form` - optional element containing data form,
- `action` - command action,
- `sessionId` - session identifier, if command is executed in session (identifier is generated by executor).

As response method returns object of `AdHocResult` class. It contains result form (optional) and status of command execution.

```
val module = halcyon.getModule<CommandsModule>(CommandsModule.TYPE)
module.executeCommand("responder@domain".toJID(), "configure").response {
    it.onSuccess { result ->
        println("Status: ${result.status}")
        println("Form: ${result.form}")
    }
}.send()
```

If this command creates session, we can simply use data from `result` to execute next command in it:

```
result.form.getFieldByVar("password").fieldValue = "1234"
module.executeCommand(result.jid, result.node, result.form.createSubmitForm(), null,␣
↪result.sessionId).response {
    it.onSuccess { result ->
        when(result.status){
            Status.Completed -> println("Configured")
            Status.Canceled -> println("Command canceled")
            Status.Executing -> println("Configuration is not finished yet. Next step in␣
↪session ${result.sessionId} is required.")
        }
    }
}.send()
```

Above example uses default action in second step (`null` on arguments list).

registerAdHocCommand(command: AdHocCommand)

In addition to executing commands on other XMPP entities, module allows to register ad-hoc commands, to be executed in client by others.

Command must implement `AdHocCommand` interface.

```
class TestAdHoc : AdHocCommand {

    override fun isAllowed(jid: BareJID): Boolean = jid == "owner@example.com".
↪toBareJID()

    override val node: String = "command-node-name"
```

(continues on next page)

(continued from previous page)

```
    override val name: String = "Example command"

    override fun process(request: AdHocRequest, response: AdHocResponse) {
        response.form = createForm()
        response.notes = arrayOf(Note.Info("Everything is OK"))
        response.status = Status.Completed
    }
}
```

Above sample command may be executed only by `owner@example.com`. It even be hidden on commands list for others.

Created command must be registered in `CommandModule`:

```
module.registerAdHocCommand(TestAdHoc())
```

Ad-hoc commands supports sessions. Session allows to store some data in session context and creates multiple stages commands. By default sessions are not started automatically. To access to session in command, use method `request.getSession()`. This method returns current session context or creates new one if it is necessary.

```
class SessionTestAdHoc : AdHocCommand {

    override fun isAllowed(jid: BareJID): Boolean = jid == "owner@example.com".
↪toBareJID()

    override val node: String = "example-session-adhoc"
    override val name: String = "Example session command"

    override fun process(request: AdHocRequest, response: AdHocResponse) {
        var counter = request.getSession().values["stage"] as Int? ?: 0
        ++counter
        request.getSession().values["stage"] = counter
        if (counter < 3) {
            response.notes = arrayOf(Note.Info("Step $counter"))
            response.actions = arrayOf(Action.Next)
            response.defaultAction = Action.Next
            response.status = Status.Executing
        } else {
            response.notes = arrayOf(Note.Info("Finished"))
            response.status = Status.Completed
        }
    }
}
```

If response status is `Completed` or `Canceled`, then session context is destroyed after command execution.

Note: Remember, that single instance of Ad-Hoc Command may process calls from many callers.

5.12 MUCModule

This module implements XEP-0045: Multi-User Chat.

5.12.1 Events

There are two kinds of events in `MUCModule`:

1. Room related events.
2. Other events

Currently the only event from second category is `InvitationReceived`:

```
halcyon.eventBus.register<MucEvents.InvitationReceived>(MucEvents.TYPE) { event ->
    println("${event.invitation.sender} invites you to room ${event.invitation.roomjid}")
}
```

All room related events contains `Room` object, and all have common parent:

```
halcyon.eventBus.register<MucRoomEvents>(MucRoomEvents.TYPE) { event ->
    when(event){
        is MucRoomEvents.YouJoined -> println("You joined to room ${event.room.roomJID}")
        is MucRoomEvents.OccupantCame -> println("Occupant ${event.nickname} came to $
↪{event.room.roomJID}")
        is MucRoomEvents.OccupantLeave -> println("Occupant ${event.nickname} leaves␣
↪room ${event.room.roomJID}")
        // ...
    }
}
```

InvitationReceived

Event fired when an invitation is received. Contains `Invitation` object.

YouJoined

Event fired when server responses for join request with success.

YouLeaved

Event fired, when you leave room. It may be confirmation of you leave request, or you are kicked out from room.

JoinError

Event fired when server not accepted join request.

Created

Event informs that room you joined is just created (by join request).

OccupantCame

Informs that new occupant joined to room.

OccupantLeave

Informs that occupant leaves the room.

OccupantChangedPresence

Informs that occupant updated his presence.

ReceivedMessage

Event fired when group chat message from room is received.

5.12.2 Methods

join(roomJID, nickname, password)

Builds join request to MUC Room.

Here is simple example to show how to join to room.

```
halcyon.eventBus.register<MucRoomEvents.YouJoined>(MucRoomEvents.TYPE) {
    println("You joined to room ${it.room.roomJID} as ${it.nickname}")
}

mucModule.join("coven@chat.shakespeare.lit".toBareJID(), "thirdwitch").send()
```

Note, that because of MUC protocol specificity, confirmation of join will be delivered as separated event.

`leave(room: Room)`

Builds request to leaves MUC Room.

`destroy(room: Room)`

Builds room destroy request.

`invite(room: Room, invitedJid: BareJID, reason: String? = null)`

Builds mediated invitation request.

`inviteDirectly(room: Room, invitedJid: BareJID, reason: String? = null)`

Builds direct invitation request.

`retrieveRoomConfig(room: Room)`

Builds retrieve room configuration request. In response it returns data form with configuration.

`updateRoomConfig(room: Room, form: JabberDataForm)`

Builds update room configuration request.

`message(room: Room, msg: String)`

Builds group chat message request.

`decline(invitation: Invitation, reason: String? = null)`

Builds decline request for received invitation

`accept(invitation: Invitation, nickname: String)`

Builds join request to MUC Room based on received invitation.

`retrieveAffiliations(room: Room, filter: Affiliation? = null)`

Builds request for retrieve affiliations list from MUC room. In response it returns collection of RoomAffiliation.

updateAffiliations(room: Room, affiliations: Collection<RoomAffiliation>)

Builds request for update affiliations list.

updateRoomSubject(room: Room, subject: String?)

Builds request for set room subject.

ping(room: Room)

Build request for self ping, as described in XEP-0410: MUC Self-Ping (Schrödinger's Chat).

5.12.3 Store

MUCModule requires Room Storage to store rooms data. By default, Halcyon comes with in-memory store.

To build own Store you have to implement this interface:

```
interface MUCStore {

    fun findRoom(roomJID: BareJID): Room?

    fun createRoom(roomJID: BareJID, nickname: String): Room
}
```

Remember, that MUC protocol is not suitable to mobile clients, so `state` of room join (in `Room` object) may not represent real state. For example, after reconnection client may keep state `Joined` but server received information about disconnection and removes occupant from room. To check if rejoin is required, please use `ping(room)` function.

5.13 SIMSModule

This module implements partially XEP-0385: Stateless Inline Media Sharing (SIMS). It describes file sharing metadata.

`SIMSModule` isn't module in Halcyon architecture sense. This is rather collection of extensions to help retrieve and generate SIMS structures.

5.13.1 Methods

getReferenceOrNull()

Method returns `Reference` object if exists.

getMediaSharingFileOrNull()

This method (extends `Reference`) returns shared file details.

createFileSharingReference()

This method creates complete `Reference` object with shared file details.

5.14 MAMModule

This module implements XEP-0313: Message Archive Management.

5.14.1 Events

MAMMessageEvent

This event is fired, when each result for `query()` method is received.

This event contains fields:

- `resultStanza` - whole received Message stanza (response for query),
- `queryId` - query identifier,
- `id` - result identifier,
- `forwardedStanza` - result for query containing stanza from archive and original receiving timestamp

5.14.2 Methods

query(to: BareJID? = null, node: String? = null, rsm: RSM.Query? = null, with: String? = null, start: Long? = null, end: Long? = null)

Main method to retrieve chat history from archive.

Arguments:

- `to` - JID of MAM component. If null, then default MAM component of user server is used,
- `node` - name of node,
- `rsm` - Result Set Management object,
- `with` - name (JID) of interlocutor
- `start`, `end` - timestamps to filter messages by receive date

All of above argoments may be `null`.

retrievePreferences()

Retrieves MAM preferences.

In response, you will get `Preferences` object containing:

- `default` - default behaviour of message archiving: `Always`, `Never`, `Roster`.
- `always` - collection of BareJIDs with whom conversations will always be archived.
- `never` - collection of BareJIDs with whom conversations will never be archived.

updatePreferences(preferences: Preferences)

Updates MAM preferences.

5.14.3 Usage

When client establish connection to server, it should ask server for all messages exchanged with other clients connected to the same account.

It can be done by asking server for all messages since, last received message:

```
val mamModule = halcyon.getModule<MAMModule>(MAMModule.TYPE)
fun ask(q: RSM.Query? = null) {
    mamModule.query(
        with = "someone@tigase.org",
        start = lastReceivedMessageTimestatmp,
        rsm = q
    ).response { res ->
        res.onSuccess {
            println("Complete: ${it.complete} :: ${it.rsm}")

            if (!it.complete) {
                ask(RSM.Query(after = it.rsm!!.last))
            }
        }
    }.consume { forwardedStanza: ForwardedStanza<Message> ->
        if (forwardedStanza.stanza.body != null) println(
            "FROM MAM | ${forwardedStanza.resultId} ${timestampToISO8601(forwardedStanza.
↪timestamp!!)} ${forwardedStanza.stanza.from}: ${forwardedStanza.stanza.body}"
        )
    }.send()
}
ask()
```

Because MAM server has defined maximum amount of returned messages, we have to ask until query is not complete. In example it is done by recurrent execution of method `ask()` with filled Result Set Management object. `RSM.Query(after = it.rsm!!.last)` means that in result must be included only messages located after last message identifier from currently received package of messages.

Note, that parameter `with` of method `query()` is optional, so you can ask for all messages since specific time exchanged with specific JID, or you can ask for all messages stored in archive.

The second way to retrieve messages from archive is asking for messages located before or after specific message identifier.

```
val mamModule = halcyon.getModule<MAMModule>(MAMModule.TYPE)
fun ask(q: RSM.Query? = null) {
    mamModule.query(
        with = "someone@tigase.org",
        rsm = q
    ).response { res ->
        res.onSuccess {
            println("Complete: ${it.complete} :: ${it.rsm}")

            if (!it.complete) {
                ask(RSM.Query(after = it.rsm!!.last))
            }
        }
    }.consume { forwardedStanza: ForwardedStanza<Message> ->
        if (forwardedStanza.stanza.body != null) println(
            "FROM MAM | ${forwardedStanza.resultId} ${timestampToISO8601(forwardedStanza.
↪timestamp!!)} ${forwardedStanza.stanza.from}: ${forwardedStanza.stanza.body}"
        )
    }.send()
}
ask(RSM(after = lastKnownMessageId))
```

To get MAM related message ID, you have to use `getStanzaIDBy()` function (As parameter, you have to put name of own account):

```
val mamMessageId = message.getStanzaIDBy("myaccount@tigase.org".toBareJID())
```

It is useful when message is received "in normal way", so when it is pushed to client by server.

When message is received as result of using method `query()` (in consumer), identifier is given in `ForwardedStanza`:

```
val mamMessageId = forwardedStanza.resultId
```

The same mechanism you can use to load older messages in history (not loaded yet by client):

```
mamModule.query(
    with = "someone@tigase.org",
    rsm = RSM.Query(before = "2753e4a8-9150-4e34-8757-4cd5e8419159", max = 20)
).response { res ->
    res.onSuccess {
        println("Complete: ${it.complete} :: ${it.rsm}")
    }
}.consume { forwardedStanza: ForwardedStanza<Message> ->
    println(
        "FROM MAM | ${forwardedStanza.resultId} ${
            timestampToISO8601(
                forwardedStanza.timestamp!!
            )
        } ${forwardedStanza.stanza.from}: ${forwardedStanza.stanza.body}"
    )
}.send()
```

In above example, client is asking for 20 messages located in history before message `2753e4a8-9150-4e34-8757-4cd5e8419159`.

Note: Message Archive returns requested amount of messages. Not every message may contain body to show. Some of those messages may contain only confirmation of message read or other controlling commands.

CHAPTER

SIX

JABBER DATA FORM

Jabber Data Form is described in XEP-0004. Data forms are useful in all workflows not described in XEPs. For example service configuration or search results.

6.1 Working with forms

To access fields of received form, we have to create `JabberDataForm` object:

```
val form = JabberDataForm(formElement)
```

Where `formElement` is representation of `<x xmlns='jabber:x:data'>` XML element.

Each form may have properties like:

- `type` - form type,
- `title` - optional title of form,
- `description` - optional, human-readable, description of form.

Fields are identified by `var` name. Each field may have field type (it is optional).

Let look, how to list all fields with values:

```
val form = JabberDataForm(element)
println("Title: ${form.title}")
println("Description: ${form.description}")
println("Type: ${form.type}")
println("Fields:")
form.getAllFields().forEach {
    println(" - ${it.fieldName}: ${it.fieldType}  (${it.fieldLabel})  == ${it.fieldValue}
↪")
}
```

To get field by name, simple use:

```
val passwordField = form.getFieldByVar("password")
```

Value of those fields may be modified:

```
passwordField.fieldValue = "******"
```

After all form modification, sometimes we need to send filled form back. There is separated method to prepare submit-ready form:

```
val formElement = form.createSubmitForm()
```

This method prepares <x xmlns='jabber:x:data'> XML element with type submit and all fields are cleared up from unnecessary elements like descriptions or labels. It just leaves simple filed with name and value.

6.2 Creating forms

We can create new form, set title and description, and add fields:

```
val form = JabberDataForm.create(FormType.Form)
form.addField("username", FieldType.TextSingle)
form.addField("password", FieldType.TextPrivate).apply {
    fieldLabel = "Enter password"
    fieldDesc = "Password must contain at least 8 characters"
    fieldRequired = true
}
```

To get XML element containing form without cleaning it, just use:

```
val formElement = form.element
```

6.3 Multi value response

There is a variant of form containing many sets of fields. This kind of form has declared set of column with names and set of items containing field with names declared before.

This example shows how to display all fields with values:

```
val form = JabberDataForm(element)
val columns = form.getReportedColumns().mapNotNull { it.fieldName }
columns.forEach { print("$it;  ") }
println()
println("------------")
form.getItems().forEach { item ->
    columns.forEach { col -> print("${item.getValue(col).fieldValue};  ") }
    println()
}
```

Creating multi value form is also simple. First we have to set list of reported columns, because when new item is added, field names are checked against declared columns.

```
val form = JabberDataForm.create(FormType.Result)
form.title = "Bot Configuration"
form.setReportedColumns(listOf(Field.create("name", null), Field.create("url", null)))
form.addItem(
    listOf(Field.create("name").apply { fieldValue = "Comune di Verona - Benvenuti nel␣
↪sito ufficiale" },
           Field.create("url").apply { fieldValue = "http://www.comune.verona.it/" })
)
form.addItem(
```

(continues on next page)

(continued from previous page)

```
    listOf(Field.create("name").apply { fieldValue = "Universita degli Studi di Verona -␣
↪Home Page" },
           Field.create("url").apply { fieldValue = "http://www.univr.it/" })
)
```