Halcyon

Release 0.1

Tigase, Inc.

Jun 21, 2023

CONTENTS

CHAPTER

ONE

WELCOME

Halcyon is an XMPP client library written in a Kotlin programming language. It provides implementation of core of the XMPP standard and processing XML. Additionally it provides support for many popular extensions (XEP's).

Library using Kotlin Multiplatform feature to provide XMPP library for as many platforms as possible. Currently we are focused on

- JVM
- JavaScript
- Android

In the future we want to provide native binary version.

CHAPTER

TWO

GETTING STARTED WITH A HALCYON

CHAPTER

THREE

SETTING UP A CLIENT

3.1 Supported platforms

Halcyon library can be used on different platforms:

- JVM
- Android
- JavaScript

3.2 Adding client dependencies

To use Halcyon library in your project you have to configure repositories and add library dependency. All versions of library are available in Tigase Maven repository:

Production

```
repositories {
    maven("https://maven-repo.tigase.org/repository/release/")
}
```

Snapshot

```
repositories {
    maven("https://maven-repo.tigase.org/repository/snapshot/")
}
```

At the end, you have to add dependency to `tigase.halcyon:halcyon-core` artifact:

```
implementation("tigase.halcyon:halcyon-core:$halcyon_version")
```

Where `$halcyon_version` is required Halcyon version.

CHAPTER

FOUR

CREATING AND CONFIGURING A CLIENT

When repositories and dependencies are configured, we can create instance of Halcyon:

```
import tigase.halcyon.core.builder.createHalcyon

val halcyon = createHalcyon {
}
```

4.1 Authentication

Of course, it requires a bit of configuration: to connect to XMPP Server, client requires username and password:

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.toBareJID

val halcyon = createHalcyon {
    auth {
        userJID = "username@xmppserver.com".toBareJID()
        password { "secretpassword" }
    }
}
```

4.2 Registering new account

To register new account on XMPP server you need separate instance of Halcyon, configured exactly for this purpose.

```
import tigase.halcyon.core.builder.createHalcyon

val halcyon = createHalcyon {
    register {
        domain = "xmppserver.com"
        registrationFormHandler { form ->
            form.getFieldByVar("username")!!.fieldValue = "username"
            form.getFieldByVar("password")!!.fieldValue = "password"
        }
    }
}
```

Note: The server may provide a different set of fields and it is the developer's responsibility to handle them.

4.3 Connectors

Halcyon library is able to use many connection methods, depends on platform. By default JVM and Android uses Socket and JavaScript uses WebSocket connector.

4.3.1 JVM SocketConnector

In Socket Connector you may configure own DNS resolver, set custom host and port, and define trust manager to check SSL server certificates.

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.builder.socketConnector

val halcyon = createHalcyon {
    socketConnector {
        dnsResolver = CustomDNSResolver()
        hostname = "127.0.0.1"
        port = 15222
        trustManager = MyTrustManager()
    }
}
```

Warning: Note that by default Halcyon doesn't check SSL server certificates at all!

4.3.2 JavaScript WebSocketConnector

If your target platform is JavaScript, then default connector will use WebSocket.

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.builder.webSocketConnector

val halcyon = createHalcyon {
    webSocketConnector {
        webSocketUrl = "ws://127.0.0.1:5290/"
    }
}
```

WebSocket connector has only one configuration parameter: server URL.

4.4 Starting and stopping

Now we are ready to connect client to the XMPP server:

```
halcyon.connectAndWait()
halcyon.disconnect()
```

Method `connectAndWait()` is JVM only method, it esteblish connection in blocking way. To start connection in async mode you have to use `connect()` method. If library was configured to register new account, thise method will start registration process. Method `disconnect()` terminates XMPP session, closes streams and sockets.

4.5 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

FIVE

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

SIX

MODULES

Architecture of Halcyon library is based on plugins (called modules). Every feature like authentication, sending and receiving messages or contact list management is implemented as module. Halcyon contains all modules in single package (at least for now), so no need to add more dependencies.

To install module you have to use `install` function:

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.modules.discovery.DiscoveryModule

val halcyon = createHalcyon {
    install(DiscoveryModule)
}
```

Most of modules can be configured. Configuration may be passed in `install` block:

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.modules.discovery.DiscoveryModule

val halcyon = createHalcyon {
    install(DiscoveryModule) {
        clientName = "My Private Bot"
        clientCategory = "client"
        clientType = "bot"
    }
}
```

By default, function `createHalcyon()` automatically add all modules. If you want to configure your own set of modules, you have to disable this feature and add required plugins by hand:

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.modules.discovery.DiscoveryModule

val halcyon = createHalcyon(installAllModules = false) {
    install(DiscoveryModule)
    install(RosterModule)
    install(PresenceModule)
}
```

Note: Despite of the name, with `install` you can also configure preinstalled modules!

Halcyon modules mechanism is implementing modules dependencies, it means that if you install module (for example) `MIXModule`, Halcyon automatically install modules `RosterModule`, `PubSubModule` and `MAMModule` with default configuration.

There is also set of aliases, to make configuration of popular modules more comfortable.

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.builder.bind

val halcyon = createHalcyon() {
    bind {
        resource = "my-little-bot"
    }
}
```

In this example we used `bind{}` instead of `install(BindModule){}`.

List of aliases:

Alias	Module name
`bind()`	`BindModule`
`sasl()`	`SASLModule`
`sasl2()`	`SASL2Module`
`discovery()`	`DiscoveryModule`
`capabilities()`	`EntityCapabilitiesModule`
`presence()`	`PresenceModule`
`roster()`	`RosterModule`

6.1 PresenceModule

Module for handling received presence information.

6.1.1 Install and configure

To install or configure preinstalled Presence module, call function `install` inside Halcyon configuration (see *Modules*):

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.modules.presence.PresenceModule
import tigase.halcyon.core.xmpp.modules.presence.InMemoryPresenceStore

val halcyon = createHalcyon {
    install(PresenceModule) {
        store = InMemoryPresenceStore()
    }
}
```

The only one configuration property `store` allows to use own implementation of presence store.

6.1.2 Setting own presence status

After connection is established, PresenceModule automatically sends initial presence.

To change own presence status, you should use *sendPresence* function.`

```
import tigase.halcyon.core.xmpp.modules.presence.PresenceModule

halcyon.getModule(PresenceModule)
    .sendPresence(
        show = Show.Chat,
        status = "I'm ready for party!"
    )
    .send()
```

It is also possible to send direct presence, only for specific recipient:

```
import tigase.halcyon.core.xmpp.modules.presence.PresenceModule

halcyon.getModule(PresenceModule)
    .sendPresence(
        jid = "mom@server.com".toJID(),
        show = Show.DnD,
        status = "I'm doing my homework!"
    )
    .send()
```

6.1.3 Presence subscription

Details of managing presence subscription are explained in XMPP specification. Here we simply show how to subscribe and unsubscribe presence with Halcyon library.

All subscriptions manipulation may be done with single *sendSubscriptionSet* function:

```
import tigase.halcyon.core.xmpp.modules.presence.PresenceModule

halcyon.getModule(PresenceModule)
    .sendSubscriptionSet(jid = "buddy@somewhere.com".toJID(), presenceType =␣
↪PresenceType.Subscribe)
    .send()
```

Depends on action you want, you have to change `presenceType` parameter:

Subscription request:
Use `PresenceType.Subscribe` to send subscription request to given JabberID.

Accepting subscription request:
Use `PresenceType.Subscribed` to accept subscription request from given JabberID.

Rejecting subscription request:
Use `PresenceType.Unsubscribed` to reject subscription request or cancelling existing subscription to our presence from given JabberID.

Unsubscribe contact:
Use `PresenceType.Unsubscribe` to cancel your subscription of given JabberID presence.

Note: Remember that subscription manipulation can affect your roster content.

6.1.4 Checking presence

When you develop application, probably you will want to check presence of your contact, to see if he is available. Halcyon provides few function for that: `getBestPresenceOf` returns presence with highest priority (in case if there are few entities under the same bare JID); `getPresenceOf` returns last received presence of given full JID. You can also check list of all entities resources logged as single bare JID with `getResources` function.

Because determining of contact presence using low-level XMPP approach is not so intuitive, we introduced `TypeAndShow`. It joins presence stanza type and show extension in single set of enums.

```
import tigase.halcyon.core.xmpp.modules.presence.PresenceModule
import tigase.halcyon.core.xmpp.modules.presence.typeAndShow

val contactStatus = halcyon.getModule(PresenceModule)
    .getBestPresenceOf("dad@server.com".toBareJID())
    .typeAndShow()
```

Thanks to it, `contactStatus` value will contain easy to show contact status like online, offline, away, etc.

6.1.5 Events

Module can fire two types of events:

- `PresenceReceivedEvent` is fired when any Presence stanza is received by client. Event contains JID of sender, stanza type (copied from stanza) and whole received stanza.
- `ContactChangeStatusEvent` is fired when received stanza changes contact presence (all subscriptions requests are ignored). Event contains JID of sender, human readable status description, current presence with highest priority and 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.

6.2 RosterModule

Module for managing roster (contact list).

6.2.1 Install and configure

To install or configure preinstalled Roster module, call function `install` inside Halcyon configuration (see *Modules*):

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.modules.roster.RosterModule
import tigase.halcyon.core.xmpp.modules.roster.InMemoryRosterStore

val halcyon = createHalcyon {
    install(RosterModule) {
        store = InMemoryRosterStore()
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The only one configuration property `store` allows to use own implementation of roster store.

6.2.2 Retrieving roster

When connection is established, client automatically requests for latest roster, so no additional actions are required.

Most modern XMPP server supports roster versioning. Thanks to it, client do not have to receive whole roster from server (which can be large). So we recommend, to implement own RosterStore to keep current roster content between client launches.

6.2.3 Manipulating roster

To add new contact to you roster you have to call `addItem` function:

```
import tigase.halcyon.core.xmpp.modules.roster.RosterModule
import tigase.halcyon.core.xmpp.modules.roster.RosterItem

halcyon.getModule(RosterModule)
    .addItem(
        RosterItem(
            jid = "contact@somewhere.com".toBareJID(),
            name = "My friend",
        )
    )
    .send()
```

Warning: Remember, that (as described in RFC) after call (and send) roster modification request, your local store will not be updated immediately. Roster store is updated only on server request!

When roster item is saved in your roster store, Halcyon fires `RosterEvent.ItemAdded` event.

To modify existing roster item, you have to call exactly the same `addItem` function:

```
import tigase.halcyon.core.xmpp.modules.roster.RosterModule
import tigase.halcyon.core.xmpp.modules.roster.RosterItem

halcyon.getModule(RosterModule)
    .addItem(
        RosterItem(
            jid = "contact@somewhere.com".toBareJID(),
            name = "My best friend!",
        )
    )
    .send()
```

The difference is that after local store update Halcyon fires `RosterEvent.ItemUpdated` event.

Last thing is removing items from roster:

```
import tigase.halcyon.core.xmpp.modules.roster.RosterModule
import tigase.halcyon.core.xmpp.modules.roster.RosterItem

halcyon.getModule(RosterModule)
    .deleteItem("contact@somewhere.com".toBareJID())
    .send()
```

When item will be removed from local store, Halcyon fires `RosterEvent.ItemRemoved` event.

6.2.4 Events

Roster module can fires few types of events:

- `RosterEvent` is fired when roster item in your local store is modified by server request. There are three sub-events: `ItemAdded`, `ItemUpdated` and `ItemRemoved`.
- `RosterLoadedEvent` inform us that roster data loading is finished. It is called only after retrieving roster on client request.
- `RosterUpdatedEvent` is fired, when processing roster data from server is finished. I will be triggered after requesting roster from server and after processing set of roster item manipulations initiated by server.

6.3 DiscoveryModule

This module implements XEP-0030: Service Discovery.

6.3.1 Install and configure

To install or configure preinstalled Discovery module, call function `install` inside Halcyon configuration (see *Modules*):

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.modules.discovery.DiscoveryModule

val halcyon = createHalcyon {
    install(DiscoveryModule) {
        clientCategory = "client"
        clientType = "console"
        clientName = "Code Snippet Demo"
        clientVersion = "1.2.3"
    }
}
```

The `DiscoveryModule` configuration is provided by interface `DiscoveryModuleConfiguration`.

- The `clientCategory` and `clientType` properties provides information about category and type of client you develop.

List of allowed values you can use is published in Service Discovery Identities document.

- The `clientName` and `clientVersion` properties contains human readable software name and version.

Note: If you change client name and version, it is good to update node name in header-EntityCapabilitiesModule.

6.3.2 Discovering information

Module provides function `info` to prepare request to get information about given entity:

```
import tigase.halcyon.core.xmpp.modules.discovery.DiscoveryModule
import tigase.halcyon.core.xmpp.toJID

halcyon.getModule(DiscoveryModule)
    .info("tigase.org".toJID())
    .response { result ->
        result.onFailure { error -> println("Error $error") }
        result.onSuccess { info ->
            println("Received info from ${info.jid}:")
            println("Features " + info.features)
            println(info.identities.joinToString { identity ->
                "${identity.name} (${identity.category}, ${identity.type})"
            })
        }
    }
    .send()
```

In case of success, module return `DiscoveryModule.Info` class containing information about requested JID and node, list of received identities and list of features.

6.3.3 Discovering list

Second feature provided by module is discovering list of items associated with an entity. It is implemented by the `items` function:

```
import tigase.halcyon.core.xmpp.modules.discovery.DiscoveryModule
import tigase.halcyon.core.xmpp.toJID

halcyon.getModule(DiscoveryModule)
    .items("tigase.org".toJID())
    .response { result ->
        result.onFailure { error -> println("Error $error") }
        result.onSuccess { items ->
            println("Received info from ${items.jid}:")
            println(items.items.joinToString { "${it.name} (${it.jid}, ${it.node})" })
        }
    }
    .send()
```

In case of success, module return `DiscoveryModule.Items` class containing information about requested JID, node and list of received items.

6.3.4 Events

After connection to server is established, module automatically requests for for features of user account and server.

When Halcyon receives account information, then `AccountFeaturesReceivedEvent` event is fired. In case of receiving XMPP server information, Halcyon fires `ServerFeaturesReceivedEvent` event.

6.4 EntityCapabilitiesModule

This module implements XEP-0115: XMPP Ping.

6.4.1 Install and configure

To install or configure preinstalled EntityCapabilities module, call function `install` inside Halcyon configuration (see *Modules*):

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.modules.caps.EntityCapabilitiesModule

val halcyon = createHalcyon {
    install(EntityCapabilitiesModule) {
        node = "http://mycompany.com/bestclientever"
        cache = MyCapsCacheImplementation()
        storeInvalid = false
    }
}
```

The `EntityCapabilitiesModule` configuration is provided by interface `EntityCapabilitiesModuleConfig`.

- The `node` is URI to identify your software. As default library uses `https://tigase.org/halcyon` URI.
- With `cache` propery you can use own implementation of capabilities cache store, for example JDBC based, to keep all received capabilities between your application restarts.
- The `storeInvalid` property allow to force storing received capabilities with invalid versification string. By default, it is set to `false`.

For more information about the possible consequences of disabling the validation verification string, refer to the Security Considerations chapter.

6.4.2 Getting capabilities

You can get entity capabilities based on the presence received.

```
import tigase.halcyon.core.xmpp.modules.caps.EntityCapabilitiesModule

val caps = halcyon.getModule(EntityCapabilitiesModule)
    .getCapabilities(presence)
```

The primary use of the module is to define a list of features of the client with whom communication is taking place. After receiving presence from client we can determine features implemented by it:

```
import tigase.halcyon.core.xmpp.modules.caps.EntityCapabilitiesModule

val caps = halcyon.getModule(EntityCapabilitiesModule)
    .getCapabilities(presence)
```

6.5 PingModule

This module implements XEP-0199: XMPP Ping.

6.5.1 Install

To install or configure preinstalled Discovery module, call function `install` inside Halcyon configuration (see *Modules*):

```
import tigase.halcyon.core.builder.createHalcyon
import tigase.halcyon.core.xmpp.modules.PingModule

val halcyon = createHalcyon {
    install(PingModule)
}
```

This module has no configuration options.

Note: If module will not be installed, other entities will not be able to ping application.

6.5.2 Pinging entity

Module provides function `ping` to prepare request for ping given entity:

```
import tigase.halcyon.core.xmpp.modules.PingModule
import tigase.halcyon.core.xmpp.toJID

halcyon.getModule(PingModule)
    .ping("tigase.org".toJID())
    .response { result ->
        result.onSuccess { pong -> println("Pong: ${pong.time}ms") }
        result.onFailure { error -> println("Error $error") }
    }
    .send()
```

In the case of success, module returns *PingModule.Pong* class containing information about measured response time.

CHAPTER

SEVEN

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:

```
import tigase.halcyon.core.xmpp.modules.PingModule

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

CHAPTER

EIGHT

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.

8.1 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.

8.2 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
```

8.3 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/" })
)
```