TigaseDoc

Release 0.1

Tigase, Inc.

Jun 23, 2023

CONTENTS

CHAPTER

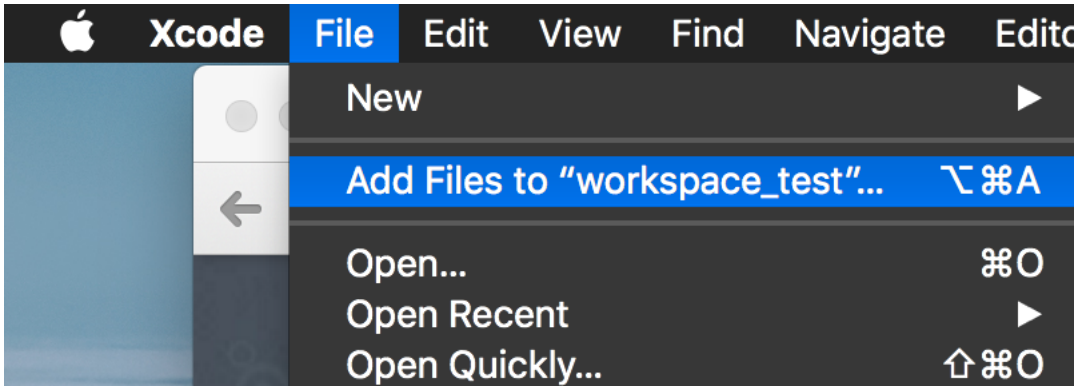
ONE

STARTING UP

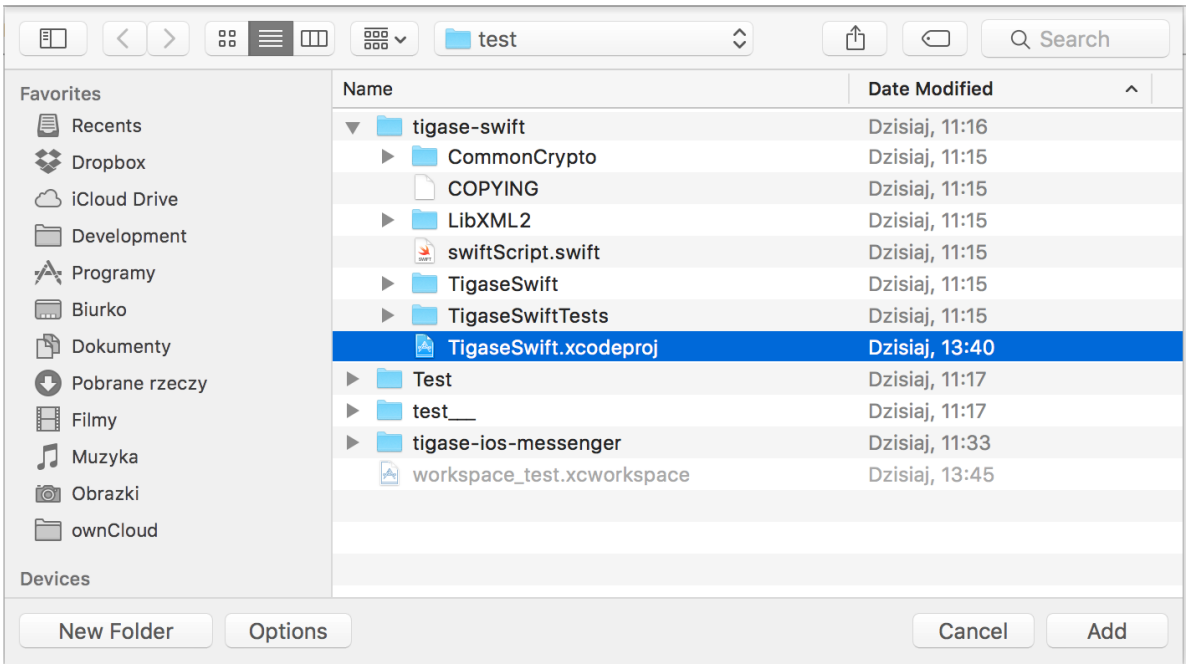
1.1 Preparation of environment for development

1. Download and install newest version of XCode
2. Download source code of library

   It is best to download library source code from our Git repository
3. Create workspace in XCode
4. Add library project to newly created workspace using `Add Files to "workspace_name"...` from `File` menu of XCode when workspace is opened in XCode.



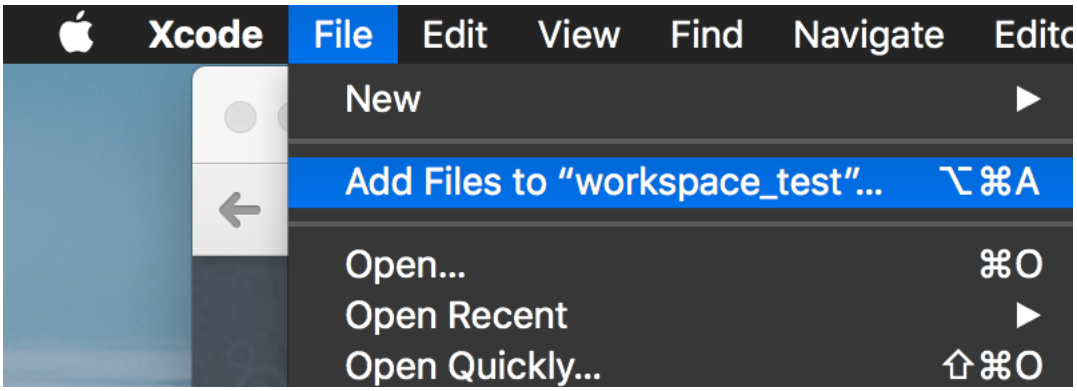
5. Select `TigaseSwift.xcodeproj` which wil be inside root directory of source code of library and click `Add`



6. Workspace for development using TigaseSwift library is ready

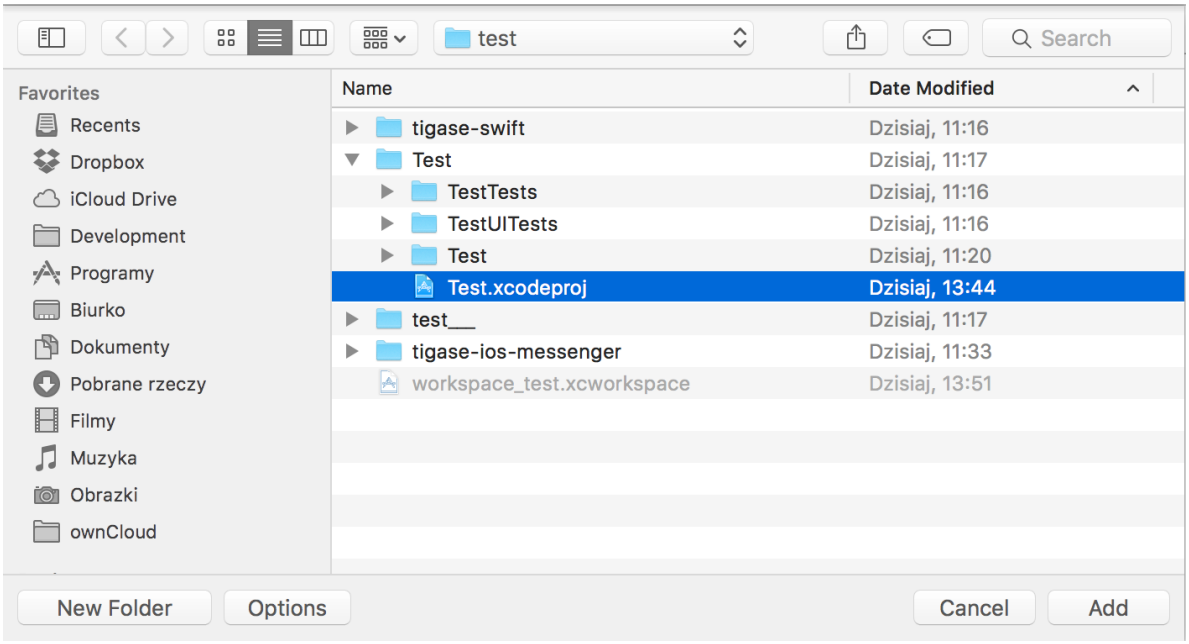
1.2 Creation of project using TigaseSwift library

1. Create project in XCode
2. Add project to TigaseSwift library workspace created during preparation of environment for development using @Add Files to "workspace_name"... @ from @File@ menu of XCode when workspace is opened in XCode.

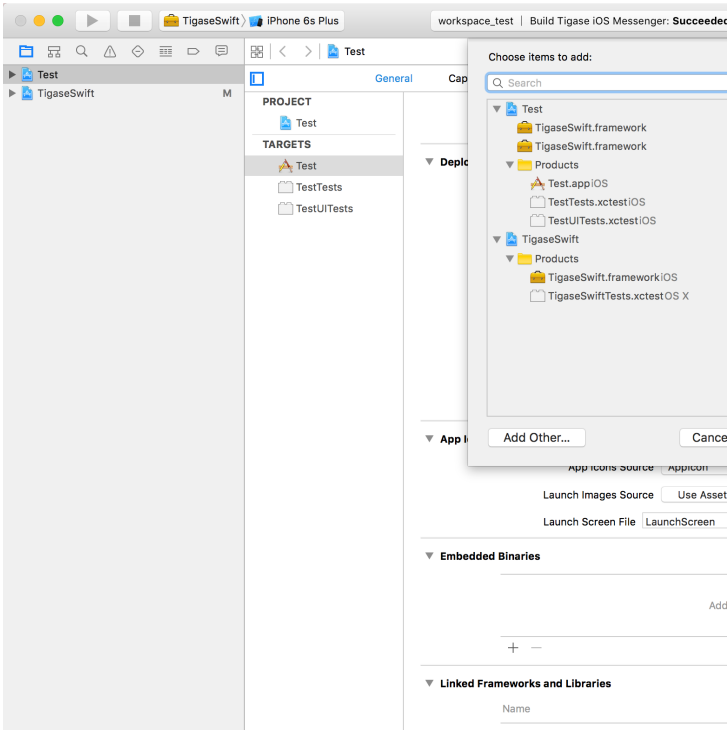


3. Select XCode project file of your newly created project and click `Add`

   This file name will end with `.xcodeproj`



4. In XCode open `Build Settings` tab of imported project
5. In section `Embedded Binaries` click on + sign
6. Select `TigaseSwift.framework` and add it to project



It will be located in `Products` folder of `TigaseSwift` project

7. Your project contains embedded `TigaseSwift` framework

CHAPTER

TWO

BASICS

2.1 Create XMPP client instance

To use TigaseSwift library you need to create instance of `XMPPClient` class which is implementation of XMPP client.

```
var client = XMPPClient();
```

2.2 Register required modules

Next step is to register modules providing support for features you would like to use. Almost in any case you will need at least following modules:

- `StreamFeaturesModule`

  Responsible for handling XMPP stream features

- `AuthModule` and `SaslModule`

  `AuthModule` add common authentication features, while `SaslModule` add support for SASL based authentication.

- `ResourceBinderModule`

  Module responsible for resource binding which is part of stream negotiation process.

- `SessionEstablishmentModule`

  Module handles session establishment which is last step of stream negotiation, however it is not needed according to RFC 6120. We recommend to register this module for compatibility reasons - if it will be not needed then it will not be used.

To register, ie. `StreamFeaturesModule` you need to use following code:

```
client.modulesManager.register(StreamFeaturesModule());
```

2.3 Register additional modules you need

You can add any additional modules found in TigaseSwift library or you can create your own based by implementing support for `XmppModule` protocol.

Here is list of some modules provided by TigaseSwift library:

- `PresenceModule`

  Responsible for handling incoming presences and allows to set client presence.

- `MessageModule`

  This module is responsible for processing incoming messages, creating/destroying chats and sending messages.

- `RosterModule`

  Provides support for retrieval and manipulation of XMPP roster.

- `MucModule`

  Provides support for MUC rooms as described in XEP-0045: Multi-User Chat

- `DiscoveryModule`

  Provides support for service discovery described in XEP-0030: Service Discovery

- `StreamManagementModule`

  Provides support for Stream Management acking and stream resumption as specified in XEP-0198: Stream Management

- `MessageCarbonsModule`

  Adds support for forwarding messages delivered to other resources as described in XEP-0280: Message Carbons

- `VCardModule`

  Implementation of support for XEP-0054: vcard-temp

- `PingModule`

  Allows to check if other XMPP client is available and it is possible to deliver packet to this XMPP client as specified in XEP-0199: XMPP Ping

- `InBandRegistrationModule`

  Adds possibility to register XMPP account using XEP-0077: In-Band Registration

- `MobileModeModule`

  Provides support for using Tigase Optimizations for mobile devices

- `CapabilitiesModule`

  Provides support for XEP-0115: Entity Capabilities which allows for advertisement and automatic discovery of features supported by other clients.

2.4 Provide credentials needed for authentication

This should be done using `connectionConfiguration` properties, ie.

```
let userJID = BareJID("user@domain.com");
client.connectionConfiguration.setUserJID(userJID);
client.connectionConfiguration.setUserPassword("Pa$$w0rd");
```

To use ANONYMOUS authentication mechanism, do not set user jid and password. Instead just set server domain:

```
client.connectionConfiguration.setDomain(domain);
```

2.5 Register for connection related events

There are three event related to connection state which should be handled:

- `SocketConnector.ConnectedEvent`

  Fired when client opens TCP connection to server - XMPP stream is not ready at this point.

- `SessionEstablishmentModule.SessionEstablishmentSuccessEvent`

  Fired when client finishes session establishment. It will be called even if `SessionEstablishmentModule` is not registered.

- `SocketConnector.DisconnectedEvent`

  Fired when TCP connection is closed or when XMPP stream is closed. It will be also called when TCP connection is broken.

2.6 Login

To start process of DNS resolution, establishing TCP connection and establishing XMPP stream you need to call:

```
client.login();
```

2.7 Disconnect

To disconnect from server properly and close XMPP and TCP connection you need to call:

```
client.disconnect();
```

2.8 Sending custom stanza

Usually class which supports `XmppModule` protocol is being implemented to add new feature to `TigaseSwift` library. However in some cases in which we want to send simple stanza or send stanza and react on received response there is no need to implement class supporting `XmppModule` protocol. Instead of that following methods may be used.

2.8.1 Sending stanza without waiting for response

To send custom stanza you need to construct this stanza and execute following code

```
client.context.writer?.write(stanza);
```

`writer` is instance of `PacketWriter` class responsible for sending stanzas from client to server. Property can be nil if connection is not established.

2.8.2 Sending stanza and waiting for response (closures)

It possible to wait for response stanza, but only in case of `Iq` stanzas. To do so, you need to pass callback which will be called when result will be received, ie.

```
client.context.writer?.write(stanza, timeout: 45, onSuccess: {(response) in
    // response received with type equal `result`
  }, onError: {(response, errorCondition) in
    // received response with type equal `error`
  }, onTimeout: {
    // no response was received in specified time
  });
```

You can omit `timeout` parameter. Default value of 30 seconds will be used as a timeout.

You can pass nil as any of closures. In this case particular response will not trigger any reaction.

2.8.3 Sending stanza and waiting for response (closure)

It possible to wait for response stanza, but only in case of `Iq` stanzas. To do so, you need to pass callback which will be called when result will be received, ie.

```
client.context.writer?.write(stanza, timeout: 45, callback: {(response) in
  // will be called on `result`, `error` or in case of timeout
  });
```

You can omit `timeout` parameter, which will use 30 seconds as default timeout.

As callback is called always as it will be called in case of received `result`, `error` or in case of timeout it is required to be able to distinguish what caused execution of this closure. In case of `result` or `error` packet being received, received stanza will be passed to closure for processing. However in case of timeout `nil` will be passed instead of stanza - as no stanza was received.

2.8.4 Sending stanza and waiting for response (AsyncCallback)

It possible to wait for response stanza, but only in case of `Iq` stanzas. To do so, you need to pass callback which will be called when result will be received, ie.

```
client.context.writer?.write(stanza, timeout: 45, callback: callback);
```

where callback is implementation of `AsyncCallback` protocol.

You can omit `timeout` parameter, which will use 30 seconds as default timeout.

CHAPTER

THREE

SUPPORTED FEATURES

Specification	Name	Description
RFC 6120	Extensible Messaging and Presence Protocol (XMPP): Core	XMPP specification including SSL/TLS encryption, SASL authentication, resource binding, etc..
RFC 6121	Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence	Roster management, presence subscription, sending and receiving messages
EP-0030	Service Discovery	Support for XMPP service discovery
XEP-0045	Multi-User Chat	Support for MUC protocol extension
XEP-0054	vcard-temp	User vCard implementation
XEP-0060	Publish-Subscribe	Support for PubSub protocol extension
XEP-0077	In-Band Registration	Support for in-band account registration
XEP-0082	XMPP Date and Time Profiles	Support for standarized ISO 8601 profiles and lexical representation
XEP-0092	Software Version	Support for discovery and advertisement of used software and it's version
XEP-0115	Entity Capabilities	Support for discovery of features supported and advertised by clients and servers
XEP-0138	Support for discovery of features supported and advertised by clients and servers	Compression of data exchanged between client and server
XEP-0153	vCard-Based Avatars	Storage of user avatar inside vCard
XEP-0163	Personal Eventing Protocol	Support for PEP protocol extension
XEP-0172	User Nickname	Support for communication of user nickname
XEP-0175	Best Practices for Use of SASL ANONYMOUS	Anonymous authentication
XEP-0198	Stream Management	
XEP-0199	XMPP Ping	
XEP-0203	Delayed Delivery	Information about delayed delivery of stanza
XEP-0280	Message Carbons	Support for delivery of messages sent to other resources
XEP-0352	Client State Indication	Notifying server about current state of XMPP client
	Mobile Optimizations	Optimizations designed for mobile devices

CHAPTER

FOUR

LIST OF SPECIFICATIONS AND MODULES IMPLEMENTING IT

RFC 6120 XMPP: Core	• AuthModule • SaslModule • StreamFeaturesModule • ResourceBinderModule • SessionEstablishmentModule
RFC 6121 XMPP: IM	• MessageModule • PresenceModule • RosterModule
XEP-0030: Service Discovery	• DiscoveryModule
XEP-0045: Multi-User Chat	• MucModule
XEP-0054: vcard-temp	• VCardModule
XEP-0060: Publish-Subscribe	• PubSubModule
XEP-0077: In-Band Registration	• InBandRegistrationModule
XEP-0092: SoftwareVersion	• SoftwareVersionModule
XEP-0115: Entity Capabilities	• CapabilitiesModule
XEP-0153: vCard-Based Avatars	• VCardModule
XEP-0163: Personal Eventing Protocol	• PubSubModule
XEP-0175: SASL Anonymous	• AnonymousMechanism
XEP-0199: XMPP Ping	• PingModule
XEP-0203: Delayed Delivery	• Delay
XEP-0280: Message Carbons	• MessageCarbonsModule
XEP-0352: Client State Indication	• ClientStateIndicationModule

CHAPTER

FIVE

USAGE EXAMPLES

5.1 Simple client sending message

Below is example code of client which send XMPP message to `recipient@domain.com` as `sender@domain.com` using `Pa$$w0rd` as password for authentication. Message is sent just after clients connects to server, authenticates and establishes session.

```
import Foundation
import TigaseSwift

class MessageSendingClient: EventHandler {

    var client: XMPPClient;

    init() {
        Log.initialize();

        client = XMPPClient();

        registerModules();

        print("Notifying event bus that we are interested in␣
↪SessionEstablishmentSuccessEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SessionEstablishmentModule.
↪SessionEstablishmentSuccessEvent.TYPE);
        print("Notifying event bus that we are interested in DisconnectedEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SocketConnector.DisconnectedEvent.
↪TYPE);

        setCredentials(userJID: "sender@domain.com", password: "Pa$$w0rd");

        print("Connecting to server..")
        client.login();
        print("Started async processing..");
    }

    func registerModules() {
        print("Registering modules required for authentication and session establishment
```

(continues on next page)

(continued from previous page)

```
↪");
        _ = client.modulesManager.register(AuthModule());
        _ = client.modulesManager.register(StreamFeaturesModule());
        _ = client.modulesManager.register(SaslModule());
        _ = client.modulesManager.register(ResourceBinderModule());
        _ = client.modulesManager.register(SessionEstablishmentModule());

        print("Registering module for sending/receiving messages..");
        _ = client.modulesManager.register(MessageModule());
    }

    func setCredentials(userJID: String, password: String) {
        let jid = BareJID(userJID);
        client.connectionConfiguration.setUserJID(jid);
        client.connectionConfiguration.setUserPassword(password);
    }

    /// Processing received events
    func handle(event: Event) {
        switch (event) {
        case is SessionEstablishmentModule.SessionEstablishmentSuccessEvent:
            sessionEstablished();
        case is SocketConnector.DisconnectedEvent:
            print("Client is disconnected.");
        default:
            print("unsupported event", event);
        }
    }

    /// Called when session is established
    func sessionEstablished() {
        print("Now we are connected to server and session is ready..");

        let messageModule: MessageModule = client.modulesManager.getModule(MessageModule.
↪ID)!;
        let recipient = JID("recipient@domain.com");
        let chat = messageModule.createChat(with: recipient);
        print("Sending message to", recipient, "..");
        _ = messageModule.sendMessage(in: chat!, body: "I'm now online..");

        print("Waiting 1 sec to ensure message is sent");
        sleep(1);
        print("Disconnecting from server..");
        client.disconnect();
    }
}
```

5.2 Simple client setting presence and handling incoming presences

In this example we will connect to server, set our presence to `Do not disturb` with status message set to `Do not disturb me!`.

This example will also print any presence we will receive from our contacts. However for this part to work our roster cannot be empty and at least one of our roster contacts needs to be available.

```
import Foundation
import TigaseSwift

class PresenceHandlingClient: EventHandler {

    var client: XMPPClient;

    init() {
        Log.initialize();

        client = XMPPClient();

        registerModules();

        print("Notifying event bus that we are interested in␣
↪SessionEstablishmentSuccessEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SessionEstablishmentModule.
↪SessionEstablishmentSuccessEvent.TYPE);
        print("Notifying event bus that we are interested in DisconnectedEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SocketConnector.DisconnectedEvent.
↪TYPE);
        print("Notifying event bus that we are interested in ContactPresenceChangedEvent
↪");
        client.eventBus.register(handler: self, for: PresenceModule.
↪ContactPresenceChanged.TYPE);

        setCredentials(userJID: "sender@domain.com", password: "Pa$$w0rd");

        print("Connecting to server..")
        client.login();
        print("Started async processing..");
    }

    func registerModules() {
        print("Registering modules required for authentication and session establishment
↪");
        _ = client.modulesManager.register(AuthModule());
        _ = client.modulesManager.register(StreamFeaturesModule());
        _ = client.modulesManager.register(SaslModule());
        _ = client.modulesManager.register(ResourceBinderModule());
        _ = client.modulesManager.register(SessionEstablishmentModule());

        print("Registering module for handling presences..");
```

(continues on next page)

(continued from previous page)

```
        _ = client.modulesManager.register(PresenceModule());
    }

    func setCredentials(userJID: String, password: String) {
        let jid = BareJID(userJID);
        client.connectionConfiguration.setUserJID(jid);
        client.connectionConfiguration.setUserPassword(password);
    }

    /// Processing received events
    func handle(event: Event) {
        switch (event) {
        case is SessionEstablishmentModule.SessionEstablishmentSuccessEvent:
            sessionEstablished();
        case is SocketConnector.DisconnectedEvent:
            print("Client is disconnected.");
        case let cpc as PresenceModule.ContactPresenceChanged:
            contactPresenceChanged(cpc);
        default:
            print("unsupported event", event);
        }
    }

    /// Called when session is established
    func sessionEstablished() {
        print("Now we are connected to server and session is ready..");

        let presenceModule: PresenceModule = client.modulesManager.
↪getModule(PresenceModule.ID)!;
        print("Setting presence to DND...");
        presenceModule.setPresence(show: Presence.Show.dnd, status: "Do not distrub me!",
↪ priority: 2);
    }

    func contactPresenceChanged(_ cpc: PresenceModule.ContactPresenceChanged) {
        print("We got notified that", cpc.presence.from, "changed presence to", cpc.
↪presence.show);
    }
}
```

5.3 Simple client setting presence, handling incoming presences and responding on incoming messages

This example presents way to listen for incoming messages and responding on this messages.

```
import Foundation
import TigaseSwift

class MessageRespondingClient: EventHandler {
```

(continues on next page)

(continued from previous page)

```
    var client: XMPPClient;

    init() {
        Log.initialize();

        client = XMPPClient();

        registerModules();

        print("Notifying event bus that we are interested in␣
↪SessionEstablishmentSuccessEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SessionEstablishmentModule.
↪SessionEstablishmentSuccessEvent.TYPE);
        print("Notifying event bus that we are interested in DisconnectedEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SocketConnector.DisconnectedEvent.
↪TYPE);
        print("Notifying event bus that we are interested in ContactPresenceChangedEvent␣
↪and in MessageReceivedEvent");
        client.eventBus.register(handler: self, for: PresenceModule.
↪ContactPresenceChanged.TYPE, MessageModule.MessageReceivedEvent.TYPE);

        setCredentials(userJID: "sender@domain.com", password: "Pa$$w0rd");

        print("Connecting to server..")
        client.login();
        print("Started async processing..");
    }

    func registerModules() {
        print("Registering modules required for authentication and session establishment
↪");
        _ = client.modulesManager.register(AuthModule());
        _ = client.modulesManager.register(StreamFeaturesModule());
        _ = client.modulesManager.register(SaslModule());
        _ = client.modulesManager.register(ResourceBinderModule());
        _ = client.modulesManager.register(SessionEstablishmentModule());

        print("Registering module for handling presences..");
        _ = client.modulesManager.register(PresenceModule());
        print("Registering module for handling messages..");
        _ = client.modulesManager.register(MessageModule());
    }

    func setCredentials(userJID: String, password: String) {
        let jid = BareJID(userJID);
        client.connectionConfiguration.setUserJID(jid);
        client.connectionConfiguration.setUserPassword(password);
    }
```

(continues on next page)

(continued from previous page)

```
    /// Processing received events
    func handle(event: Event) {
        switch (event) {
        case is SessionEstablishmentModule.SessionEstablishmentSuccessEvent:
            sessionEstablished();
        case is SocketConnector.DisconnectedEvent:
            print("Client is disconnected.");
        case let cpc as PresenceModule.ContactPresenceChanged:
            contactPresenceChanged(cpc);
        case let mr as MessageModule.MessageReceivedEvent:
            messageReceived(mr);
        default:
            print("unsupported event", event);
        }
    }

    /// Called when session is established
    func sessionEstablished() {
        print("Now we are connected to server and session is ready..");

        let presenceModule: PresenceModule = client.modulesManager.
↪getModule(PresenceModule.ID)!;
        print("Setting presence to DND...");
        presenceModule.setPresence(show: Presence.Show.dnd, status: "Do not distrub me!",
↪ priority: 2);
    }

    func contactPresenceChanged(_ cpc: PresenceModule.ContactPresenceChanged) {
        print("We got notified that", cpc.presence.from, "changed presence to", cpc.
↪presence.show);
    }

    func messageReceived(_ mr: MessageModule.MessageReceivedEvent) {
        print("Received new message from", mr.message.from, "with text", mr.message.
↪body);

        let messageModule: MessageModule = client.modulesManager.getModule(MessageModule.
↪ID)!;
        print("Creating chat instance if it was not received..");
        let chat = mr.chat ?? messageModule.createChat(with: mr.message.from!);
        print("Sending response..");
        _ = messageModule.sendMessage(in: chat!, body: "Message in response to: " + (mr.
↪message.body ?? ""));
    }
}
```

5.4 Simple client with support for MUC

In this example you can find how to join to room, send message to room and handle informations about occupants.

```
import Foundation
import TigaseSwift

class MucClient: EventHandler {

    var client: XMPPClient;

    init() {
        Log.initialize();

        client = XMPPClient();

        registerModules();

        print("Notifying event bus that we are interested in␣
↪SessionEstablishmentSuccessEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SessionEstablishmentModule.
↪SessionEstablishmentSuccessEvent.TYPE);
        print("Notifying event bus that we are interested in DisconnectedEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SocketConnector.DisconnectedEvent.
↪TYPE);

        print("Notifying event but that we are interested in some of MucModule events");
        client.eventBus.register(handler: self, for: MucModule.YouJoinedEvent.TYPE,␣
↪MucModule.MessageReceivedEvent.TYPE, MucModule.OccupantComesEvent.TYPE, MucModule.
↪OccupantLeavedEvent.TYPE, MucModule.OccupantChangedPresenceEvent.TYPE);

        setCredentials(userJID: "sender@domain.com", password: "Pa$$w0rd");

        print("Connecting to server..")
        client.login();
        print("Started async processing..");
    }

    func registerModules() {
        print("Registering modules required for authentication and session establishment
↪");
        _ = client.modulesManager.register(AuthModule());
        _ = client.modulesManager.register(StreamFeaturesModule());
        _ = client.modulesManager.register(SaslModule());
        _ = client.modulesManager.register(ResourceBinderModule());
        _ = client.modulesManager.register(SessionEstablishmentModule());

        print("Registering module for handling presences..");
        _ = client.modulesManager.register(PresenceModule());
        print("Registering module for handling messages..");
        _ = client.modulesManager.register(MessageModule());
```

(continues on next page)

(continued from previous page)

```
        print("Registering module for handling MUC...");
        _ = client.modulesManager.register(MucModule());
    }

    func setCredentials(userJID: String, password: String) {
        let jid = BareJID(userJID);
        client.connectionConfiguration.setUserJID(jid);
        client.connectionConfiguration.setUserPassword(password);
    }

    /// Processing received events
    func handle(event: Event) {
        switch (event) {
        case is SessionEstablishmentModule.SessionEstablishmentSuccessEvent:
            sessionEstablished();
        case is SocketConnector.DisconnectedEvent:
            print("Client is disconnected.");
        case let cpc as PresenceModule.ContactPresenceChanged:
            contactPresenceChanged(cpc);
        case let mr as MessageModule.MessageReceivedEvent:
            messageReceived(mr);
        case let mrj as MucModule.YouJoinedEvent:
            mucRoomJoined(mrj);
        case let mmr as MucModule.MessageReceivedEvent:
            mucMessageReceived(mmr);
        case let mro as MucModule.OccupantComesEvent:
            print("Occupant", mro.occupant.nickname, "entered room with presence", mro.
↪presence);
        case let mro as MucModule.OccupantLeavedEvent:
            print("Occupant", mro.occupant.nickname, "left room");
        case let mro as MucModule.OccupantChangedPresenceEvent:
            print("Occupant", mro.occupant.nickname, "changed presence to", mro.presence)
        default:
            print("unsupported event", event);
        }
    }

    /// Called when session is established
    func sessionEstablished() {
        print("Now we are connected to server and session is ready..");

        let presenceModule: PresenceModule = client.modulesManager.
↪getModule(PresenceModule.ID)!;
        print("Setting presence to DND...");
        presenceModule.setPresence(show: Presence.Show.dnd, status: "Do not distrub me!",
↪ priority: 2);

        let mucModule: MucModule = client.modulesManager.getModule(MucModule.ID)!;
        _ = mucModule.join(roomName: "room-name", mucServer: "muc.domain.com", nickname:
↪"Test");
    }
```

(continues on next page)

(continued from previous page)

```
    func contactPresenceChanged(_ cpc: PresenceModule.ContactPresenceChanged) {
        print("We got notified that", cpc.presence.from, "changed presence to", cpc.
↪presence.show);
    }

    func messageReceived(_ mr: MessageModule.MessageReceivedEvent) {
        print("Received new message from", mr.message.from, "with text", mr.message.
↪body);

        let messageModule: MessageModule = client.modulesManager.getModule(MessageModule.
↪ID)!;
        print("Creating chat instance if it was not received..");
        let chat = mr.chat ?? messageModule.createChat(with: mr.message.from!);
        print("Sending response..");
        _ = messageModule.sendMessage(in: chat!, body: "Message in response to: " + (mr.
↪message.body ?? ""));
    }

    func mucRoomJoined(_ event: MucModule.YouJoinedEvent) {
        event.room.sendMessage("Welcome to all");
    }

    func mucMessageReceived(_ event: MucModule.MessageReceivedEvent) {
        print("received from", event.nickname, "message", event.message.body);
    }
}
```

5.5 Simple client with support for PubSub

In this example you can find how to create PubSub node, publish item, receive notifications, retrieve items and delete PubSub node.

```
import Foundation
import TigaseSwift

class PubSubClient: EventHandler {

    var client: XMPPClient;
    var pubsubJid: BareJID!;
    let nodeName = "test-node1";

    var errorHandler: ((ErrorCondition?,PubSubErrorCondition?)->Void)? = {␣
↪(errorCondition,pubsubErrorCondition) in
        print("received error: ", errorCondition, pubsubErrorCondition);
    };

    init() {
        Log.initialize();

        client = XMPPClient();
```

(continues on next page)

(continued from previous page)

```
        registerModules();

        print("Notifying event bus that we are interested in␣
↪SessionEstablishmentSuccessEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SessionEstablishmentModule.
↪SessionEstablishmentSuccessEvent.TYPE);
        print("Notifying event bus that we are interested in DisconnectedEvent" +
            " which is fired after client is connected");
        client.eventBus.register(handler: self, for: SocketConnector.DisconnectedEvent.
↪TYPE);
        print("Notifying event bus that we are interested in ContactPresenceChangedEvent
↪");
        client.eventBus.register(handler: self, for: PresenceModule.
↪ContactPresenceChanged.TYPE);
        print("Notifying event bus that we are interedted in PubSubModule.
↪NotificationReceivedEvent");
        client.eventBus.register(handler: self, for: PubSubModule.
↪NotificationReceivedEvent.TYPE)

        setCredentials(userJID: "sender@domain.com", password: "Pa$$w0rd");

        pubsubJid = BareJID("pubsub." + client.sessionObject.userBareJid!.domain);

        print("Connecting to server..")
        client.login();
        print("Started async processing..");
    }

    func registerModules() {
        print("Registering modules required for authentication and session establishment
↪");
        _ = client.modulesManager.register(AuthModule());
        _ = client.modulesManager.register(StreamFeaturesModule());
        _ = client.modulesManager.register(SaslModule());
        _ = client.modulesManager.register(ResourceBinderModule());
        _ = client.modulesManager.register(SessionEstablishmentModule());

        print("Registering module for handling presences..");
        _ = client.modulesManager.register(PresenceModule());
        print("Registering module for handling messages..");
        _ = client.modulesManager.register(MessageModule());
        print("Registering module for handling pubsub protocol..");
        _ = client.modulesManager.register(PubSubModule());
}

    func setCredentials(userJID: String, password: String) {
        let jid = BareJID(userJID);
        client.connectionConfiguration.setUserJID(jid);
        client.connectionConfiguration.setUserPassword(password);
    }
```

(continues on next page)

(continued from previous page)

```
    /// Processing received events
    func handle(event: Event) {
        switch (event) {
        case is SessionEstablishmentModule.SessionEstablishmentSuccessEvent:
            sessionEstablished();
        case is SocketConnector.DisconnectedEvent:
            print("Client is disconnected.");
        case let cpc as PresenceModule.ContactPresenceChanged:
            contactPresenceChanged(cpc);
        case let psne as PubSubModule.NotificationReceivedEvent:
            pubsubNotificationReceived(psne);
        default:
            print("unsupported event", event);
        }
    }

    /// Called when session is established
    func sessionEstablished() {
        print("Now we are connected to server and session is ready..");

        let presenceModule: PresenceModule = client.modulesManager.
↪getModule(PresenceModule.ID)!;
        print("Setting presence to DND...");
        presenceModule.setPresence(show: Presence.Show.dnd, status: "Do not distrub me!",
↪ priority: 2);
        self.createPubSubNode();
    }

    func contactPresenceChanged(_ cpc: PresenceModule.ContactPresenceChanged) {
        print("We got notified that", cpc.presence.from, "changed presence to", cpc.
↪presence.show);
    }

    func createPubSubNode() {
        let pubsubModule: PubSubModule = client.modulesManager.getModule(PubSubModule.
↪ID)!;
        pubsubModule.createNode(at: pubsubJid, node: nodeName, onSuccess: { (stanza) in
            print("node", self.nodeName, "created at", self.pubsubJid);
            self.publishItem();
        }, onError: self.errorHandler);
    }

    func publishItem() {
        let pubsubModule: PubSubModule = client.modulesManager.getModule(PubSubModule.
↪ID)!;

        let payload = Element(name: "payload", cdata: "Sample item");

        pubsubModule.publishItem(at: pubsubJid, to: nodeName, payload: payload,␣
↪onSuccess: { (stanza,node,id) in
            print("published item with id", id, "on node", node, "at", self.pubsubJid);
```

(continues on next page)

(continued from previous page)

```
            self.retrieveItems();
        }, onError: self.errorHandler);
    }

    func retrieveItems() {
        let pubsubModule: PubSubModule = client.modulesManager.getModule(PubSubModule.
↪ID)!;

        pubsubModule.retriveItems(from: pubsubJid, for: nodeName, onSuccess: { (stanza,␣
↪node, items, rsm) in
            print("retrieved", items.count, " items from", stanza.from, "node", node,
↪"items = ", items);
            self.deletePubSubNode()
        }, onError: self.errorHandler);
    }

    func deletePubSubNode() {
        let pubsubModule: PubSubModule = client.modulesManager.getModule(PubSubModule.
↪ID)!;
        pubsubModule.deleteNode(from: pubsubJid, node: nodeName, onSuccess: { (stanza) in
            print("node", self.nodeName, "deleted from", self.pubsubJid);
        }, onError: self.errorHandler);
    }

    func pubsubNotificationReceived(_ event: PubSubModule.NotificationReceivedEvent) {
        print("received notification event from pubsub node", event.nodeName, "at",␣
↪event.message.from, "action", event.itemType, "with item id", event.itemId, "and␣
↪payload", event.payload?.stringValue);
    }
}
```