

Building a Command-Line Chat App with Node.js and Socket.IO

Full Name: TRẦN ĐÌNH NAM DƯƠNG

Class: MINF20

Subject: IOT – TD

Project: Building a Command-Line Chat App with Node.js and Socket.IO

Source code: https://github.com/dtran-hub/iot_td

Contents

1. Introduction	2
2. Purpose and Scope.....	2
3. What are Node.js and Socket.io.....	2
4. Environment setup.....	3
4.1 Code editor.	3
4.2 Node.js.	3
5. Initialization step.....	3
6. Development.....	3
6.1 First program.....	3
6.2 Namespaces	6
6.3 Store information exchange on chat	7
6.4 Unit test	8
6.5 Security enhancement.	10

1. Introduction

This document describes the process of the development of a Command-Line chat application using Node.js and Socket.IO. The purpose, the scope, the use case, and milestone of the applications.

2. Purpose and Scope.

The purpose of the chat application is to allow users being able to the chat with each other, or group via command-line, like a normal chat application.

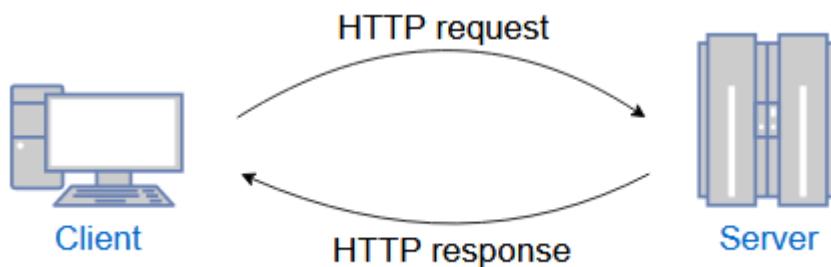
3. What are Node.js and Socket.io

- Node.js:

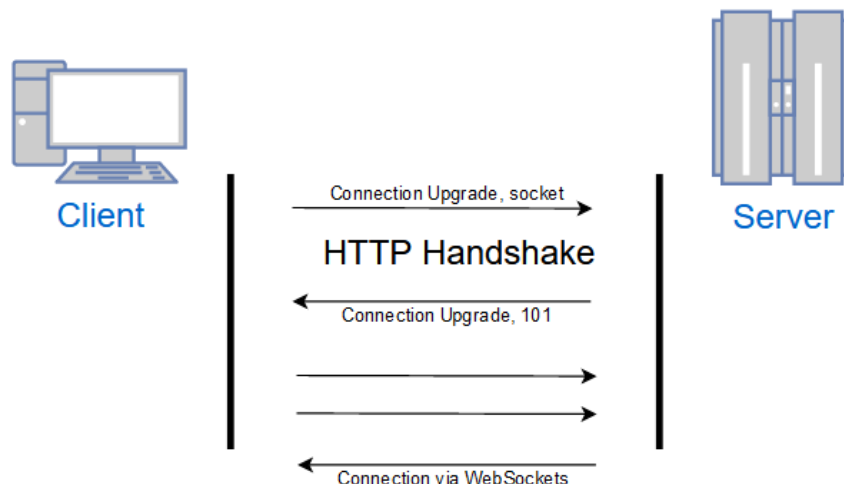
Node.js is an open source, cross-platform runtime environment for developing server end and networking applications. Node.js application is written in JavaScript, and capable run within the node.js runtime OS X, Window and Linux. There are plenty of great features that make Node.js the first choice of software architecture but the most important advantage that we can use JavaScript as both front-end and back-end language.

- Socket.io:

Socket.io is a JavaScript library for Realtime web application. It enables real time, bi-directional communication between web clients and servers.



WebSocket protocol is a protocol of the application layer of the OSI model, which enables to communicate in full duplex between client and a web server. In other words, it allows to cerate real-time web applications, like instant messaging chat.



4. Environment setup.

4.1 Code editor.

In this program we use Visual Studio Code (VSC) to write the program integrated to GitLab to publishing code to Instructor.

Version: 1.57.1 (user setup)

OS: Windows NT x64 10.0.18363

4.2 Node.js.

Node.js is download at <https://nodejs.org/en/download/>, Window installer edition, this includes Node Package Management (npm) ver. 6.14.13

```
PS F:\MyOneDrive\OneDrive\MINF20\6.a IoT\SioChat\siochat> node -v
v14.17.0
PS F:\MyOneDrive\OneDrive\MINF20\6.a IoT\SioChat\siochat> npm -v
6.14.13
```

5. Initialization step.

There are 02 newly folders created name Server and Client. We will then create the Package.JSON which contains a manifest of our project that includes the packages and applications it depends on, information about its unique source control, and specific metadata like the project's name, description, and author.

Now we initiate the project by running command `npm init -yes`. It has done as a short captured below:

```
PS F:\MyOneDrive\OneDrive\MINF20\6.a IoT\SioChat\siochat> npm init -yes
Wrote to F:\MyOneDrive\OneDrive\MINF20\6.a IoT\SioChat\siochat\package.json:

{
  "name": "siochat",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://gitlab.com/dtran-hub/siochat.git"
  },
  "keywords": [],
```

6. Development.

6.1 First program

In this section we will create a simple chat program server.js and client with

In the server directory we create a new file name server.js listening request at port 3000

```
const port = 3000;
const io = require("socket.io")(port);
console.log("Server is listening on port: %d", port);
```

Now, we create "connect" function to server respond on the event of client request. As explained above the Socket.IO is based on events, so we setup events with its handle, and when the event is emitted, the corresponding handler run.

```
io.on("connect", (socket) => {
  // Display a event when there's a user connected
  console.log("A user connected");
  socket.on("disconnect", (reason) => {
    console.log("A client disconnected, reason: %s", reason);
    console.log("Number of clients: %d", io.engine.clientsCount);
  });
});
```

A client.js program is created under folder [Client] as following to connect to the server listening on port 3000

```
client > JS client.js
1 const io = require("socket.io-client"); 47K (gzipped: 14.9K)
2 const socket = io("http://localhost:3000");
```

02 events created with corresponding handler.

Connect: Show info user connected to the chat server

```
socket.on("connect", () => {
  nickname = process.argv[2];
  console.log("[INFO]: Welcome %s", nickname);
  socket.emit("join", {"sender": nickname, "action": "join"});
});
```

Disconnect: Event is fired right after a user disconnected from the chat

```
socket.on("disconnect", (reason) => {
  console.log("[INFO]: Server disconnected, reason: %s", reason);
});
```

Program runs as follow:

Server is started and listening on port 3000

```
PS F:\MyOneDrive\OneDrive\Win2016\IoT\SioChat\siochat> cd server
PS F:\MyOneDrive\OneDrive\Win2016\IoT\SioChat\siochat\server> node .\server.js
Server is listening on port: 3000
```

A client connecting to server. It was successful with a message notified “user connected” on server-end

```
PS F:\MyOneDrive\OneDrive\Win2016\IoT\SioChat\siochat\client> node.exe .\client.js
Connecting to the server...
[INFO]: Welcome undefined

PS F:\MyOneDrive\OneDrive\Win2016\IoT\SioChat\siochat\server> node .\server.js
Server is listening on port: 3000
A user connected
Server is listening on connected to users namespace
Nickname: undefined , ID: Ern2nf8FFsiZhM8nAAAB
Number of clients: 1
```

At the first program, nickname has not yet defined but getting socket.ID to show as proof of a connection.

Implement broadcast function:

Now we implement <broadcast> function. Below spec that we will use

Command	Event	Json format	Comments
b; hello	broadcast	{sender: sender_name , msg: message_content , action: 'broadcast'}	sender_name broadcasts the message message_content

- **command:** syntax we use to send message to everyone. Here we use command syntax “b;” to send message to everyone.
- **event:** name of event sends to server.
- **Json format:** this field contains the payload that will be sent with the broadcast event. While data can be sent in several forms, JSON is simplest.
- **Readline:** a module in Node.js allow to read the input stream by the line. *Readline* function is used in this case which allows us to read the input stream line by line.

Program runs:

Client side

```
PS F:\MyOneDrive\OneDrive\WINF20\6.a IoT\SioChat\siochat\client> node.e
xe .\client.js
Connecting to the server...
[INFO]: Welcome undefined
> b;hello i am Duong
b; hello i am Duong
```

Sever side

```
Nickname: undefined , ID: Ern2nf8FFsiZhM8nAAAB
Number of clients: 1
{ action: 'broadcast', msg: 'hello i am Duong' }
```

Notification

Now we implement notification that is to communicate to everyone in chat server that a user has joined with the following Json format is used.

Command	Event	Json format	Comments
	join	{sender: sender_name , action: 'join'}	sender_name joins the chat

Program run:

```
PS F:\MyOneDrive\OneDrive\WINF20\6.a IoT\SioChat\siochat\client> node .\client.js
Connecting to the server...
[INFO]: Welcome undefined
> b;hello everyone, i just joined.
b; hello everyone, i just joined.
```

user Duong received message notified by server telling new user joined

```
> b;hello i am Duong
b; hello i am Duong
> [INFO]: undefined has joined the chat
hello i am Duong
[INFO]: undefined has joined the chat
hello everyone, i just joined.
```

Additional simple functions to enrich the program: “list” and “quit”

Here command syntax “ls;” and “q;” to send to server response for corresponding action list user and quit the chat respectively:

Command	Event	Json format	Comments
ls;	list	{sender: sender_name , action: 'list'}	sender_name lists the connected clients

Command	Event	Json format	Comments
q;	quit	{sender: sender_name , action: 'quit'}	sender_name quit the server

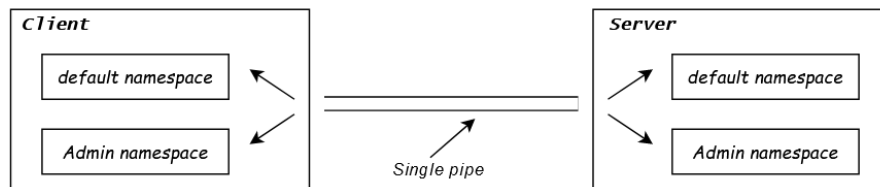
Let run and see the result:

```
[INFO]: Welcome undefined
> b;hello everyone, i just joined.
b; hello everyone, i just joined.
> ls;
[INFO]: List of nicknames:
null
null
```

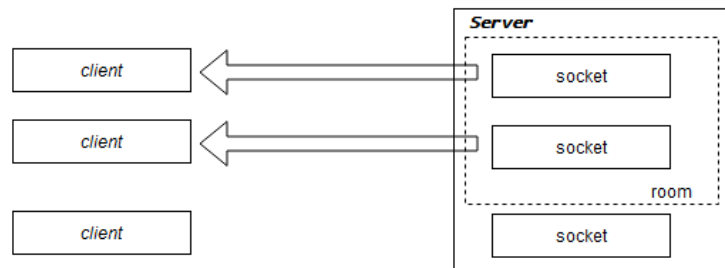
```
q;
[INFO]: Server disconnected, reason: io server disconnect
```

6.2 Namespaces

In this section, we will implement additional communication channel that allows use to separate the login of our application over a single shared connection. Common practice is to separate admin concerns from other regular users.



Chat room: Now we create a chat room using NameSpace which support to define any channels as many as our application needs that sockets being able to join and leave. We noticed that the room can be only joined on the server-side.



Now let see what we have implemented with NameSpace following the given specification from Instructor.

Command	Event	Json format	Comments
s; roni ; hello	send	{sender: sender_name , receiver: receiver_name , msg: message_content , action: 'send'}	sender_name sends the message message_content to receiver_name
jg; doctors	join_group	{sender: sender_name , group: group_name , action: 'join_group'}	sender_name joins the group group_name
bg; doctors ; hello	broadcast_group	{sender: sender_name , group: group_name , msg: message_content , action: 'broadcast_group'}	sender_name broadcasts the message message_content in the group group_name
mbr; doctors	list_members_group	{sender: sender_name , group: group_name , action: 'list_members_group'}	sender_name lists all clients that are inside the group group_name
msg; doctors	list_messages_group	{sender: sender_name , group: group_name , action: 'list_messages_group'}	sender_name lists the history of messages exchanged in the group group_name
grp;	list_groups	{name: sender_name , action: 'list_groups'}	sender_name lists the existing groups
lg; doctors	leave_group	{sender: sender_name , group: group_name , action: 'leave_group'}	sender_name leaves the group group_name

Join a group.

Message show I joined group chat [Sale] successful.

```
Message received from join room: { group: 'Sale', action: 'join_group' }
Joined the room as Sale successfully
```

Broadcast in a group

We've post a broadcast message in Chat Sale

```
Message received from join room: { group: 'Sale', action: 'join_group' }
Joined the room as Sale successfully
Message received from broadcast room: { group: 'Sale', msg: 'Hello everyone', action: 'broadcast_group' }
```

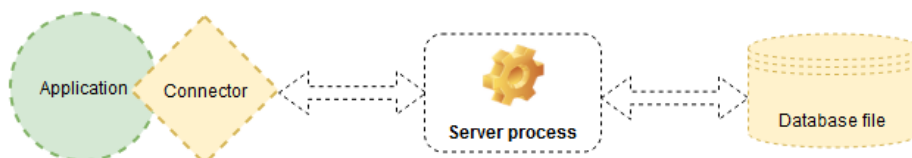
List all member in the group: By using the syntax "mbr;" to discover list of group's members

Listing group: List all groups existing the chat server

6.3 Store information exchange on chat

As programs developed previously, we store chat information as well as message exchange in chat room directly on the servers. Let imagine what will happen if the server stops or chat service crash. All information will be lost obviously as they store in main memory. Now we will improve the messages function to store all information thanks to support of SQLite as chosen.

SQLite database is integrated with the application that accesses the database. The applications interact with the SQLite database read and write directly from the database files stored on disk. The following diagram illustrates the SQLite server-less architecture:



Now we implement new program creating newly database for the chat app.

```
Chat database

var create_connection_sqlite = function () {
    Complexity is 3 Everything is cool!
    db = new sqlite3.Database('./db/chat.db', sqlite3.OPEN_READWRITE, (err) => {
        if (err) {
            console.log("Error on the DB creation:");
            return console.error(err.message);
        }
        console.log('Connected to chat.db');
    });
};

Function to store message to db

var strsqlinsert = function (room, sender, receiver, msg) {
    "strsqlinsert": Unknown
    let ret = 0;
    create_connection_sqlite();
    Complexity is 3 Everything is cool!
    db.run('INSERT INTO RoomMessage(Room, Sender, Receiver, Message) VALUES(?, ?, ?, ?)',
        if (err) {
            console.log("Exception for inserting Room Message table:");
            return console.log(err.message);
        }
        console.log('message has been inserted');
```


Function to query message from db

```

var strsqlquery = function (sql, params, subject, event, socket) {
  create_connection_sqlite();
  console.log("Everything is cool!");
  db.all(sql, params, (err, rows) => {
    if (err) {
      console.log("Error occurred when querying data on DB:");
      throw err;
    }
    rows.forEach((row) => {
      console.log(`Message kept in DB ${sql}`);
      console.log(row);
    });
  });
}

```

Now we then improve the “list_message_group” function to store chat log into the DB.

Function to list message from db

```

socket.on("list_message_group_db", (data) => {
  console.log("List message group: %s", data);
  console.log("Messages as following:");
  console.log(arrUserRoom);

  let params = [];
  let sql = "select Room, Sender, Receiver, Message from RoomMessage where Room = ?";
  params.push(data.group);
  strsqlquery(sql, params, data.group, "List message group on DB ", socket);
});

```

Let see how it works.

I simulated by running 2 clients connected to server and joined same group of Sale.

Server side

```

John has information:
[
  { Room: 'hello', Sender: 'John', Receiver: 'Sale', Message: null },
  {
    Room: 'Sale',
    Sender: 'John',
    Receiver: 'hello Sale team',
    Message: null
  }
]

```

On client's terminals

```

PS F:\MyOneDrive\OneDrive\MINF20\6.a IoT\SioChat\siochat\client> node .\client.js
Connecting to the server...
[INFO]: Welcome Duong
[INFO]: John has joined the chat
jg;Sale
Duong has joined Sale successfully
> ch;Sale;Hello Sale team
Messages from John of Sale
{
  sender: 'John',
  group: 'Sale',
  receiver: 'hello Sale team',
}
> ch;Sale;hello Sale team
> msg;
History of messages of a user in room from sqlite
[
  { Room: 'hello', Sender: 'John', Receiver: 'Sale', Message: null },
  {
    Room: 'Sale',
    Sender: 'John',
    Receiver: 'hello Sale team',
    Message: null
  }
]

```

6.4 Unit test

In this section, we describe how to manage and verify the server behaviors on each operation. Precisely, here are what we implement:

- In normal chat
 - Notify that a user joined the chat
 - Broadcast a message to others in the chat
 - List of all users in the chat

- Notify that a user quite the chat
- Send a private message to a user
- In group chat
 - Notify that a user joined a group
 - Broadcast a message to a group
 - List all users that are in a group
 - List existing groups

Now we install *mocha* and *chai as instructor*

```
+ mocha@9.0.2
added 74 packages from 44 contributors and audited 221 packages in 33.725s
```

```
+ chai@4.3.4
added 7 packages from 20 contributors and audited 228 packages in 5.347s
```

We implement `beforeeach()` and `aftereach()` functions to have a test of program. Given usernames are [Ted] and [Roni] is used to verify the program

```
describe('---Testing server in the chat', function() {
  var socket_1, socket_2;
  var username_1 = "ted";
  var username_2 = "roni"; // "roni": Unknown word.
  var options = { "force new connection": true };
  Complexity is 3 Everything is cool!
  beforeEach(function (done) {
    // This would set a timeout of 3000ms only for this hook
    this.timeout(3000);
    console.log(">> Test #" + (num_tests++));
    socket_1 = io("http://localhost:3000", options);
    socket_2 = io("http://localhost:3000", options);
    socket_1.on("connect", function() {
      console.log("socket_1 connected");
      socket_1.emit("join", { "sender": username_1, "action": "join" });
    });
    socket_2.on("connect", function() {
      console.log("socket_2 connected");
      socket_2.emit("join", { "sender": username_2, "action": "join" });
    });
  });

  afterEach(function (done) {
    // This would set a timeout of 2000ms only for this hook
    this.timeout(2000);
    socket_1.on("disconnect", function() {
      console.log("socket_1 disconnected");
    });
    socket_2.on("disconnect", function() {
      console.log("socket_2 disconnected\n");
    });
    socket_1.disconnect();
    socket_2.disconnect();
    setTimeout(done, 500); // Call done() function after 500ms
  });
});
```

Continue, we implement first 2 various function to check the chat login function and broad cast message:

```
Complexity is 9 it's time to do something...
it('Notify that a user joined the chat', function(done) {
  socket_1.emit("join", { "sender": username_1, "action": "join" });
  socket_2.on("join", function(data) {
    expect(data.sender).to.equal(username_1);
    done();
  });
});

it('Broadcast a message to others in the chat', function(done) {
  var msg_hello = "hello socket_2";
  socket_1.emit("broadcast", { "sender": username_1, "action": "broadcast", "msg": msg_hello });
  socket_2.on("broadcast", function(data) {
    expect(data.msg).to.equal(msg_hello);
    done();
  });
});
```

6.5 Security enhancement.

Finally, when everything works properly. it is time now to think about enhancement of security for this chat application which are considered one of most important part of an application, in context of the current cyber security situation. Here, we firstly improve the authentication mechanism to secure user credential on the chat.

In order to connect to the chat server, user should have to follow specification on [join] event:

Direction	Command	Event	Json format	Comments
Client -> Server		join	{sender: sender_name, password: password, action: "join"}	sender_name joins the chat with the password password

Here we use bcrypt module to hash password. Hashing by bcrypt module combined with salts protects user against rainbows table attacks.

First, bcrypt module is installed.

```
+ bcrypt@5.0.1
added 56 packages from 110 contributors and audited 195 packages in 20.563s
```

Next, it is implemented to the program as below:

```
const port = 3000;
const io = require("socket.io")(port);
const sqlite3 = require('sqlite3').verbose();
const bcrypt = require('bcrypt');
```

include the bcrypt module in the program

Bcrypt provides both asynchronous and synchronus password hashing methods. Here we use `bcrypt.hash()` function takes in the plain password and a salt as input, and returns a hashed string, as below:

```
if (rows.length == 0) {
  Complexity is 5 it's time to do something...
  bcrypt.hash(pwd, saltRounds, function (subErr, hash) {
    if (subErr) {
      console.log("Error occurred on bcrypt hash function");
      return console.log(subErr);
    }
  });
}
```

Use bcrypt.hash to encrypt password

Implement check function to verify a login

```
socket.on("join", (data) => {
  let sql = "select Username, Password from Accounts where Username = ?";
  let username = data.sender;
  let pwd = data.password;
  var filterUserArr = arrUser.filter(user => user.name != username);
  arrUser = filterUserArr;
  arrUser.push({
    name: username,
    id: socket.id
  });
});
```

Query user and password from DB

Let see how it works when there's a new login

```
PS F:\MyOneDrive\OneDrive\MINF20\6.a IoT\SioChat\siochat\client> node .\client.js Peter 123456
Connecting to the server...
(node:26620) [DEP0106] DeprecationWarning: crypto.createCipher is deprecated.
(Use `node --trace-deprecation ...` to show where the warning was created)
[INFO]: Welcome Peter
> █
```

And on server side

```
Close the database connection.
Connected to the SQLite database.
A row has been inserted with rowid 1 in Accounts table
Close the database connection.
Access Granted!

Nickname: Peter , ID: b0evS625DxDGKVP_AAAB
Number of clients: 1
```

Message encryption: It's time now to enhance the message transmit over the network

On client side, we implement a hash function to encrypt message advance of sending by using the CryptoCipherKey function

```
socket.on("private message with crypto", (data) => {
  var msg = cryptoCipherKey.update(data.msg, 'hex', 'utf8');
  msg += cryptoDecipherKey.final('utf8');
  console.log("%s send you private message with crypto: %s", data.sender, msg);
});
```

With the command "cryptmsg;" to send private message requiring encryption

```
} else if (true === input.startsWith("cryptmsg;")) { "cryptmsg": Unknown word.
  tempArr = input.split(';');
  var msg = cryptoCipherKey.update(tempArr[2], 'utf8', 'hex');
  msg += cryptoCipherKey.final('hex');
  socket.emit("private message with crypto", {
    "sender": nickname,
    "receiver": tempArr[1],
    "msg": msg,
    "action": "send_crypto"
  });
};
```

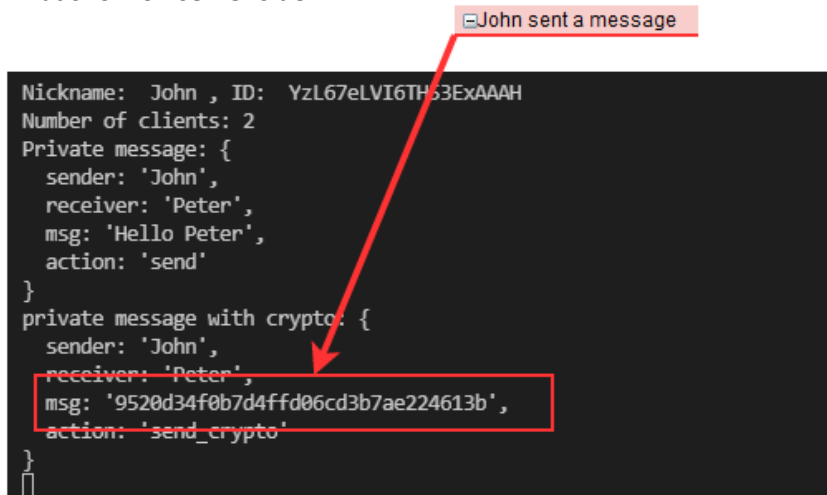
Let run program and see how it work. we take an example that user John will be sending a private message to Peter and requires to encrypt over the transmission.

```
PS F:\MyOneDrive\OneDrive\MINF20\6.a IoT\SioChat\siochat\client> node .\client.js John 123456
Connecting to the server...
(node:14940) [DEP0106] DeprecationWarning: crypto.createCipher is deprecated.
(Use `node --trace-deprecation ...` to show where the warning was created)
[INFO]: Welcome John
> s;Peter;Hello Pete
r
> cryptmsg;Peter;Hello Peter
```

Here what show on server side:

John sent a message

```
Nickname: John , ID: YzL67eLVI6TH53ExAAAH
Number of clients: 2
Private message: {
  sender: 'John',
  receiver: 'Peter',
  msg: 'Hello Peter',
  action: 'send'
}
private message with crypto: {
  sender: 'John',
  receiver: 'Peter',
  msg: '9520d34f0b7d4ffd06cd3b7ae224613b',
  action: 'send_crypto'
}
```



And what Peter received

Peter got decrypted message from John

```
(use node --trace-deprecation ... to show where the warning was created)
[INFO]: Welcome Peter
> [INFO]: John has joined the chat
> John send you private message: Hello Peter
John send you private message with crypto: Hello Peter
> 
```

