My Community Chat Changelog

This Document will guide you to implement the new event messages.

Rename Conversation

A new event for renaming a Group Chat or a normal chat.

- Event Name: `chat.renameConversation`
- Data Required:
  - `conversationId: string`
  - `conversationName: string`
- Example:

```
const data = {
    conversationId: '5acfd5af27dd78376bdda5f5',
    conversationName: `Friends for Ever`
};
socket.emit('chat.renameConversation', data);
```

this will rename the conversation to `Friends for Ever` and when you list your conversations you should expect a `conversationName` property on conversation payload.

Join Conversation `aka add users to conversation`

A new event for creating a Group Chat

- Event Name: `chat.joinConversation`
- Data Required:
  - `conversationId: string`
  - `userIds: string[]`
- Example:

```
const data = {
    conversationId: '5acfd5af27dd78376bdda5f5',
    userIds: ['30', '32']
};
socket.emit('chat.joinConversation', data);
```

this will add user `30` and `32` to the chat conversation `5acfd5af27dd78376bdda5f5`

Leave Conversation `aka remove users from conversation`

> Same as joining

- Event Name: `chat.leaveConversation`
- Data Required:
  - `conversationId: string`
  - `userIds: string[]`
- Example:

```
const data = {
    conversationId: '5acfd5af27dd78376bdda5f5',
    userIds: ['30', '32']
};
socket.emit('chat.leaveConversation', data);
```

this will remove user `30` and `32` from the chat conversation `5acfd5af27dd78376bdda5f5`

Get Conversation Messages

- Event Name: `chat.conversationMessages`
- Data Required:
  - `conversationId: string`
  - `since: string | ISODateString` - *optional*, *default*= 7 Days ago -
  - `limit: number` - *optional* , *default*= 25 -
- Example:

```
const data = {
    conversationId: '5acfd5af27dd78376bdda5f5',
    limit: 30,
    since: '2018-05-05 01:12:38.297Z' ,
};
socket.emit('chat.conversationMessages', data);
```

> see more code here

After emitting this event you should get the last messages from this conversation created since the provided date at `since` property.

> TL;DR: the `createdAt` prop, it is a Date Type format in ISO Date String, see more about ISO Date here in MongoDB.

```
{
    messages: [
    {
        content:"Ping !",
        conversationId:"5ada0cadfc62bc4afeeb5af3",
        createdAt:"2018-05-06T05:20:07.589Z",
        from:"2",
        hasMedia:false,
        mediaIds:[],
        mediaType:[],
        mediaUrls:[],
        status:2,
        recipients:["30"],
        updatedAt:"2018-05-06T05:20:07.589Z",
        __v:0,
        _id:"5aee9087532f6354e87098e1", // this the id of the message
    },
    ...
    ]
}
```

you also could expect that could be an empty array `[]` so at this time thats means there is no new message since you viewed this conversation.

so how to query the messages ? or how to paginate !
it is now easy, at first, you should cache messages, at least, the last couple of messages, then when you open the chat layout, you should read the `createdAt` property from the last message you have in the cache, then send it to the server with this event.
the server will lookup and see if there is more message at the given `conversationId` since this time - *the time of the last message you have* - and then the server will send you back an array of these messages if any.

this technique solved two problems until now, first the undelivered messages, since we could now get the new messages that we haven't received in our cache, so it's easy and lightweight on the server and on the client to achieve this, the second problem was the huge payload and bandwidth usage, early we used to send a fixed size of messages `25` normally, and the server was not able to decide if we get this message before or not, so it was a duplication of work in booth the client and the server.

Message Seen/Delivered Event

The New Message Seen Event has been added

- Event Name: `chat.messageSeen`
- Data Required:
    - `conversationId: string`
    - `users: string[]`
    - `messageIds: string[]`
- Example:

```
const data = {
    conversationId: '5acfd5af27dd78376bdda5f5',
    users: ['1', '30'],
    messageIds: ['5ade79900665dd546645da7c','5ade6af97d3a36715baa889d'],
};
socket.emit('chat.messageSeen', data);
```

> see more code here

After emitting this event, the server will update the status of message to be marked as `Seen`.
But what kind of message status flags ?
Currently we have 3 flags as mentioned in the following `enum`

```
enum MessageStatus {
    NOT_DELIVERED = 0,
    DELIVERED, // = 1
    SEEN, // = 2
}
```

then the server will emitting new update to all current users in the conversation, with new updates on this event
`chat.conversationUpdated`

the new update will be the same data sended, the server will give u the new message Ids that have be marked as `seen`

```
...
// the data object is the object that received
// from the client
object: {
   conversationId: data.conversationId,
   messageIds: data.messageIds,
   status: MessageStatus.SEEN, // => 2
};
```

User Typing Event

The New Typing Event has been added

- Event Name: `chat.typing`
- Data Required:
  - `conversationId: string`
  - `users: string[]`
- Example:

```
const data = {
    conversationId: '5acfd5af27dd78376bdda5f5',
    users: ['1', '30'],
};
socket.emit('chat.typing', data);
```

> see more code [here]

when you emit this event, the server will send updates to all clients in this conversation, so they know who is typing.
the response will take this shape

```
...
object :{
    conversationId: '5acfd5af27dd78376bdda5f5',
    userTyping: sender.id // e.g => '30'
}
...
```

the update will be received in the same event name `chat.typing`.

> BONUS: see how `debounceEvent` works [here]()

we will standardize the `timeout` of `debounceEvent` to equal `1500` ms.

User Online Event

The New User Online Event has been added.

There is 2 events here to handle online users.

1. Get User Online Friends.
2. Notify My Followers That I'm Active.

what? Friends ! Followers ? what the heck the followers and friends?

Let me explain each of this concepts:
in server there is 3 type of Relationships between users.

- Contact: an entity means that user is in my contacts list, but that dose not mean that contact is a user.
- Friend: an entity means that contact is in my contacts list, also this contact has an active user on our system.
- Follower: an entity means that someone added me on his contacts list, but that's by mean that i'm also added this user in my contacts.

Example:
We have users: `A`, `B` and `C`
User `A` added in his contacts list 2 new contacts `B` And `C`.
The System will check for these new 2 Contacts.
let's assume that user `B` has an active user in our system, but user `C` not.
The Relationship will be :

```
- User `B` ⇒  User `A` = Friend & Contact.
- User `C` ⇒ User `A` = Contact.
- User `A` ⇒ User `B` = Follower.
```

Ok let's continue with our events.

1. Get User Online Friends

- Event Name: `chat.checkOnlineFriends`
- Data Required: `null`

The Response should be an Array of online users that every object will be something like this:

```
{
    isFavourite: boolean,
    contactName: string,
    isUser: true,
    userId: string,
    mobileNumber: string,
    user: User
}
```

The response will be on the same event name.

> NOTE: this response will only contain the Online Friends.

2. Notify My Followers That I'm Active.

- Event Name: `chat.iamActive`
- Data Required:
    - `iamActive: boolean`

when you emit this event, the server will get all of your followers and then emit on the same event name that user has been online/offline according to `iamActive` property value, so you should also listen on the same event name.

when some of your friends becomes online/offline, you will get data like this.

```
{


    id:  string; // the user id
    username:  string; // ignore
    email:  string; //ignore
    mobileNumber:  string;
    iamActive: boolean;
    serverId:  string; //ignore
    clientId:  string; //ignore

}
```

simply when the user going out of the scope, close application, going away for a while or whatever, just emit this event with `iamActive: false` to let other followers know that you are not available for now, then once again when the user open the application send it again with the `true` value.

> This event also open doors to implement or add option in client side to control the privacy of being online/offline, so if the user disallow to send his activity to other followers, then simply don't emit this event, and to be fair, just don't listen to it also.

Appendix

A - All Events names :

```

// Auth Events
AUTHENTICATE:  'auth.authenticate',
AUTHENTICATED:  'auth.authenticated',

// Chat Events
CREATE_CONVERSATION:  'chat.createConversation',
LEAVE_CONVERSATION:  'chat.leaveConversation',
JOIN_CONVERSATION:  'chat.joinConversation',
CONVERSATION_CREATED:  'chat.conversationCreated',
RENAME_CONVERSATION:  'chat.renameConversation',
SEND_MESSAGE:  'chat.sendMessage',
LIST_CONVERSATIONS:  'chat.listConversations',
CONVERSATION_MESSAGES:  'chat.conversationMessages',
MESSAGE_SEEN:  'chat.messageSeen',
LIST_UNDELIVERED_MESSAGES:  'chat.listUnDeliveredMessages',
RECEIVE_MESSAGE:  'chat.receiveMessage',
ACK_EVENT:  'chat.acknowledgementEvent',
TYPING:  'chat.typing',
BLOCKED:  'chat.blocked',
CONVERSATION_UPDATED:  'chat.conversationUpdated',
CHECK_ONLINE_FRIENDS:  'chat.checkOnlineFriends',
IAM_ACTIVE:  'chat.iamActive',

// Timeline Events `aka Turbofan`
FANOUT_STATUS:  'timeline.fanoutStatus',
FANIN_STATUS:  'timeline.faninStatus',

// Error Events - for error handling -
EXCEPTION_EVENT:  'app.serverException',
INTERNAL_SERVER_ERROR:  'app.serverInternalError',
```

Questions ?