SFChat v0.11.0 technical specification

by SFChat Team

Objective

SFChat is acronym for Secure Free Chat under BSD-3-Clause license.

Technology

BackEnd:

- Python 3.4
- Django 1.7 framework
- Django REST Framework

- MongoDB
- Python-RQ
- Comet long polling
- Chat protocol
- Rest api
- SSL/TLS

FrontEnd:

- JQuery framework
- Comet long polling
- Chat class for handling users events
- Session Storage for conversation history

Protocol

Overview

Chat protocol should:

1. Minimize saving information on Server
2. Only two person can be speak through SFChat
3. Chat can be active 1 day

Investigation

There are two more famous chat protocols: XMPP (Extensible Messaging and Presence Protocol) and IRC (Internet Relay Chat). All that protocols are complicated and illustrate how interlocutors connect with each other by using pull of servers with secure and safe way. They are solved multi users and multi server collisions. But SFChat does not has such problems because only one server and only two users are involved to message change process. Therefore it's efficient to create new protocol for SFChat.

Glossary

Term	**Definition**
SFChat server	SFChat server provides connection between chat participants. It works like a phone or internet provider do.
invitation code	Unique identifier alphanumeric string with length 24 characters. It used to identify chat. Such identifier is _id in MongoDB chat collection. So "invitation code" have to sent through secure channel e.g mobile phone, email, etc.
system message	Type of message that is sent to participant by SFChat to notice about chat status, errors, etc.
long pulling	FrontEnd technique to reduce number of request to the server

chat creator	Person who creates new chat and send “invitation code” for another participant
chat joiner	Person who join to chat conversation using “invitation code”
chat token	The same value as a “invitation code”
user token	Unique identifier of user. Such identifier generates during authorization process. It helps identify user and chat. It’s generated by “ObjectId()” in MongoDB.
message token	Unique identifier of message. It’s used for handle message delivery process. It’s generated by “ObjectId()” in MongoDB.

Introduction

Protocol arrange message conversation between two users.
Each chat has 4 phases:

1. Create chat
2. Join chat
3. Conversation
4. Exit chat

Create chat

Workflow has those steps:

1. new instance of chat is added to database
2. “invitation code” is generated
3. system message was registered for “chat creator” with “invitation code”,
4. “user token” was created

Join chat by code

Join chat process contains of:

1. “invitation code” verifies
2. chat status was set as “ready”
3. “user token” was created
4. generate system message that chat is ready

Conversation

Conversation exchange has such essentials components:

- “Message” send to SFChat server
- “Long pulling” get new messages and displays on a chat
- If message was successfully sent to addresser then it would be removed from database

Exit

Exit or close chat has such phases:

- For user that click to end conversation is:
    - chat marked with “closed” status
    - “system message” for all users is created that chat was closed
    - conversation history removed from Session Storage
    - when user refresh chat page with “close” status then they will be redirected to main page
- For another user:
    - “Long pulling” get “system message” that chat was closed
    - “Success delivery confirmation” starts to clear Database from chat information
    - FrontEnd “long polling” requests are stopped
    - conversation history is removed from Session Storage

Messages

There are two type of messages “user” and “system”.

System messages

The main distinguish between messages that “system”:

- Bordered with asterix
- Used as information for chat participants
- Can be displayed only for one user

System messages shows as a result of chat events:

- New chat was created and “invitation code“ (please read more about code in corresponding section) was generated
- Uses joins to chat
- Chat was closed by participant
- Error appears (validation, internal chat errors etc.)

Message has such structure:

- Border on the top and button with asterix
- Message text

User messages

User message has more complicate structure than “system” message. It contains:

- Time in [H:i:s] format
- Participant name: “You”, “Talker” or “...”. Name “...” is used if several messages from one user was sent
- Message body

For more information about style please follow to mockup section.

History

Message history is saved on FrontEnd in Session Storage. Message history is removed during “exit” process.

Semantic structure

Semantic structure for “user message” is:

```
<div id=”msg-%message token%” class=”message”>
    <div class=”msg-date”>13:22:51</div>
    <div class=”msg-name”>You</div>
    <div class=”msg-text”>Hi, how are you doing today?</div>
</div>
```

Semantic structure of “system message” is:

```
<div id=”msg-%message token%” class=”message system”>
    <div class=”border”></div>
    <div class=”msg-text”>Internal error has occured. Your message will be resend automatically again, please contact to help desk if such error appears again.</div>
    <div class=”border”></div>
</div>
```

Where %message token% is placeholder for “message token”.

Note: “message token” is reserved for future usage to indicate that message is on a delivery way. It’s just idea.

Chat Statuses

Status shows in what state particular entity now. Workflow is different accordingly status. So please follow corresponding workflow section for more details.

Name	Description
draft	New chat was created, “invitation code” was generated. Chat is “wait” for join another participant.
ready	So chat is ready for secure use. “Invitation code” was accepted.
closed	Chat was closed with one of participant. It helps to block sending new messages and remove all history from database.

Rest api

Communication between SFChat server and FrontEnd arranges by REST api with JSON data format.

Endpoint

Endpoint is /api/

Versioning

For versioning it is used url ex. /api/v1/ or header X-SFC-VERSION=v1

URL Example

http://127.0.0.1:8000/api/v1/messages.json

where:

- http://127.0.0.1:8000 - host name
- /api - endpoint
- v1 version - number
- messages - resources
- .json - datatype

Authorization

Authorization process has vary from “chat creator” and “chat joiner”.
“Chat creator”:

1. creates chat,
2. generates “Invitation code” and saves it in database,
3. generates “user token” and saves it in database.

“Chat joiner”:

1. verify “Invitation code”
2. set chat status as “ready”
3. generate “user token” and save it in database.

For details please follow diagram.

Authentication

Authentication process works through “user token” and “chat token”. Each Api request has such required parameters:

- “chat token” that identify chat
- “user token” that helps to identify user

For details please follow diagram.

Resources

Each resources have general required header’s parameters:

Headers	Type	Description
X-SFC-userToken	String[24]	Unique key for authenticate user in chat
X-SFC-chatToken	String[24]	Chat Token unique chat identifier

For instance for messages resource development environment url looks like:
http://127.0.0.1:8000/api/v1/messages.json

GET: messages.json

Return list of new messages. Such resource is used for “long polling”.

Request

Optional parameter that turn on/off long-polling:

Parameter	Type	Default	Description
longPolling	Boolean	True	Default value True

Response

Table below describes parameters in response body:

Parameters	Type	Description
results	Object	root object for response
results.code	Integer	response code
results.msg	String	response message
results.count	Integer	number of messages
results.status	Boolean	chat status
results.messages	Array	container of messages
results.messages._id	String[24]	message identifier
results.messages.msg	String[140]	message body
results.messages.system	Boolean	true if message is system or false otherwise

Response code is: 200 if Okey or other errors code otherwise.

For instance response with two messages would look like:

```
{
    'results': {
        'code': 200,
        'msg': 'Ok',
        'count': 2,
        'status': 'ready'
        'messages': [
```

```
                0: {
                        '_id':'0cbc6611f5540bd0809a388dc95a615b',
                        'msg': 'Hi, how are you?',
                        'system': false
                },
                1: {
                        'token':'0cbc6611f5540bd0809a388dc95a615a,
                        'msg': 'Where are you?',
                        'system': false
                }
        ]
    }
}
```

Empty response has that structure:

```
{
    'results': {
        'code': 200,
        'msg': 'Ok',
        'count': 0,
        'status': 'ready'
        'messages': [
        ]
    }
}
```

POST: messages.json

Send new message on SFChat server.

Request

Request does not have any special parameters. But body contains data with in structure that describes below.

Parameters	Type	Description
data	Object	root object
data.messages	Array	root element for messages
data.messages.msg	String[140]	message body

Note: It's possible that for some reason data were not sent so to prevent missing message the FrontEnd client keeps sending message until max attempt limit will be reach.

Example of body is:

```
{
        'data': {
                messages : [
                        0: {
                                'msg': 'Message body'
                        }
                ]
        }
}
```

Response

Response is standard.

DELETE: messages.json

Delete messages from SFChat. It's used generally as a result of successfully delivery messages. In this case SFChat server keeps only undelivered messages.

Request

Request does not have any special parameters.

Parameters	Type	Description
data	Object	root object
data.messages	Array	root element for messages
data.messages._id	String[24]	message identifier

Example of body is:

```
{
        'data': {
                messages : [
                        0: {
                                '_id': '0cbc6611f5540bd0809a388dc95a615b'
                        }
                ]
        }
}
```

Response

Response is standard.

DELETE: chat.json

Delete chat means that one of participants want to exit from conversation. Therefore all undelivered messages should be delivered and chat should completely removed from SFChat server.

Request

Don’t have any special parameters.

Response

Response is standard.

Standard response

Parameters	Type	Description
results	Object	root object
results.code	Integer	response code
results.msg	String	response message

Response code is: 202 if Okey or other errors code otherwise.

Example:

```
{
    'results': {
        'code': 200,
        'msg': 'Ok'
    }
}
```

Note: for “error” code like 500, 404, etc. message should be displayed as a “system message”. For full list of used http codes please follow the corresponding section.

Http codes

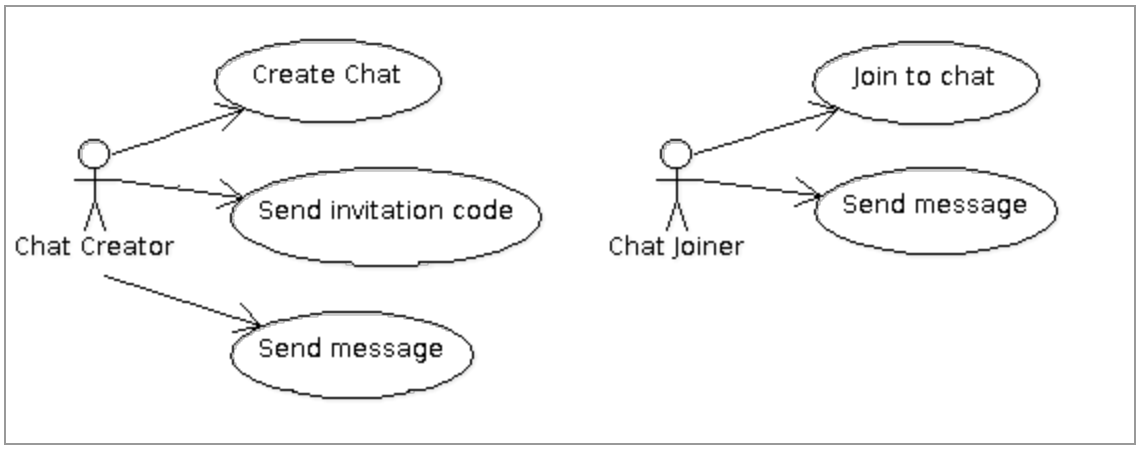
Code	Message	Standard description	SFChat description
200	Ok	The request has succeeded.	For all successful request GET
202	Accepted	The request has been accepted for processing, but the processing has not been completed.	Sending new message to SFChat server. But it is not guarantee that message will be delivery to addressee
401	Unauthorized	The request requires user authentication.	User token is invalid
404	Not Found	The server has not found anything matching the Request-URI.	Undefined resource

501	Not Implemented	The server does not support the functionality required to fulfill the request.	Method for such resource is not allowed
500	Internal Server Error	The server encountered an unexpected condition which prevented it from fulfilling the request.	Unexpected situation was happen. That makes error, exception, etc.

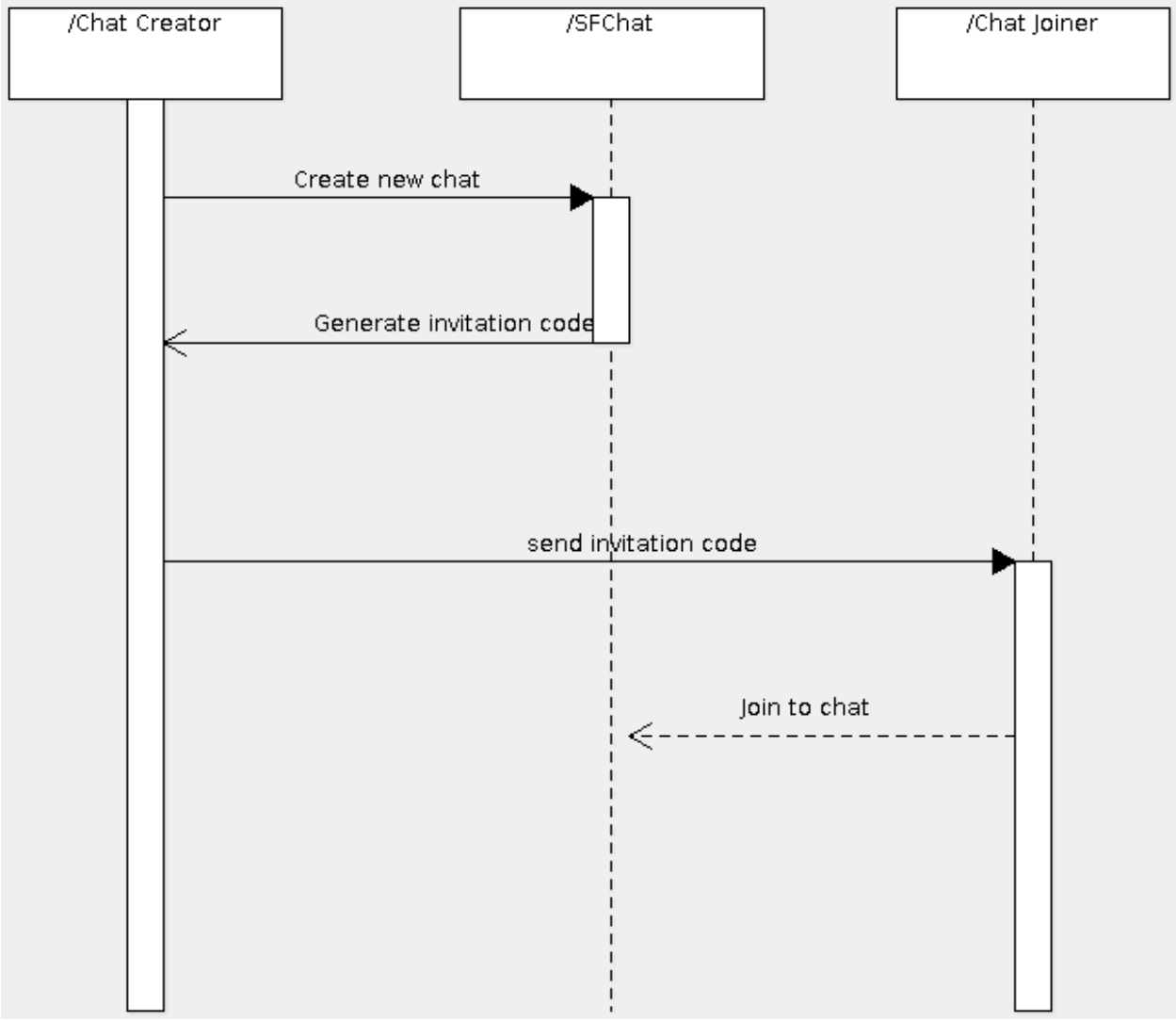
Note: Rest API Framework is used all set of http errors handlers so table above is not a full list of probably codes.

Diagrams

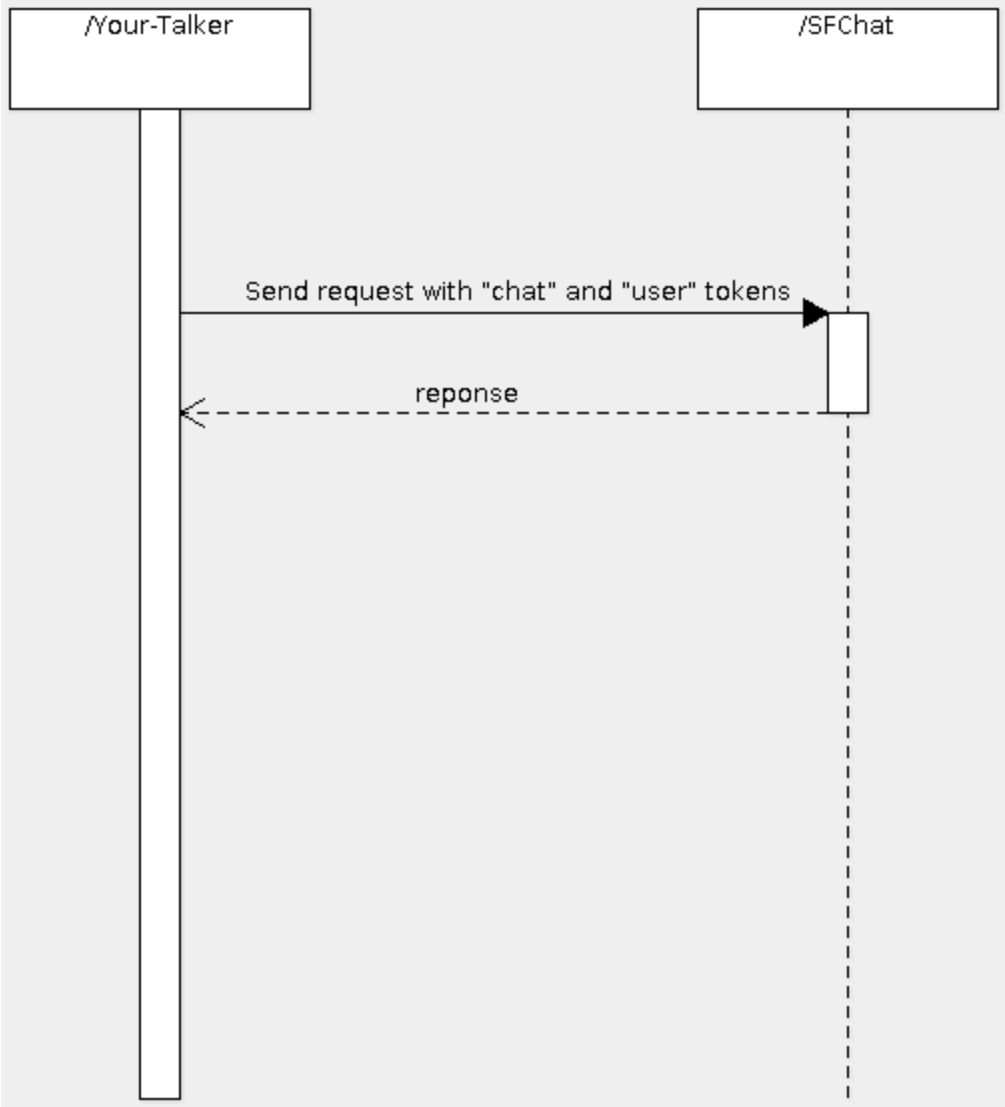
Use case diagram



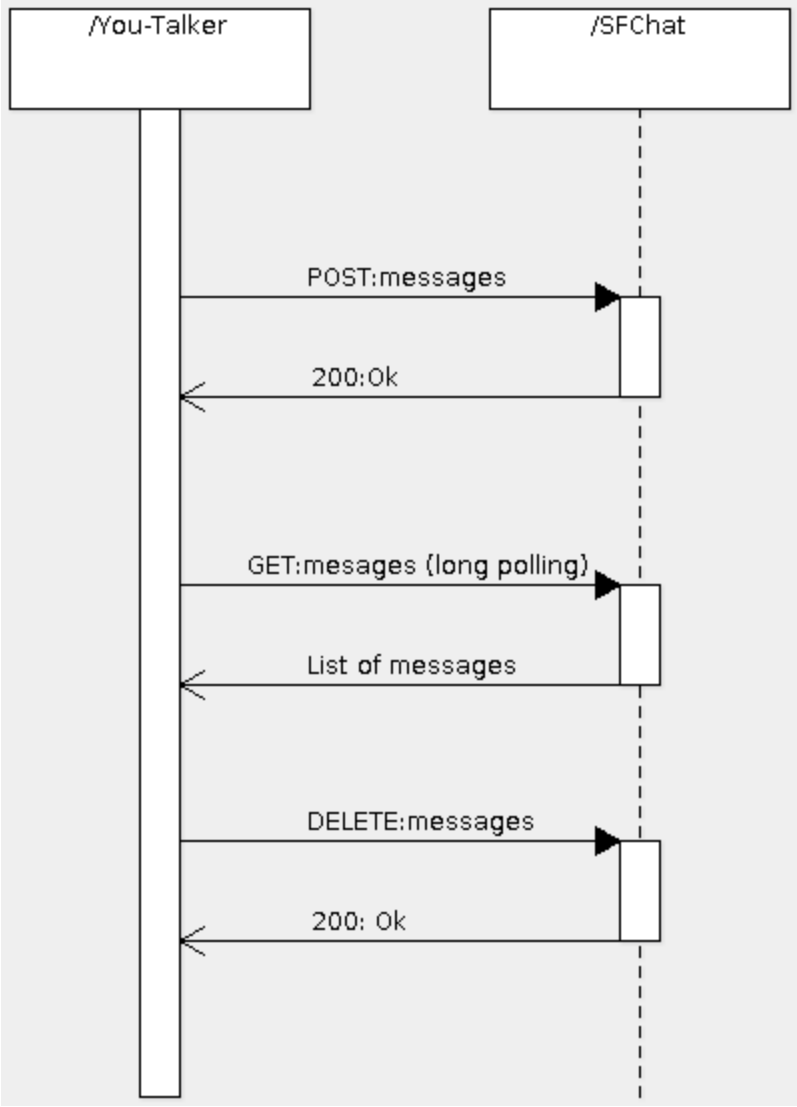
Sequence diagram authorization (create-join)



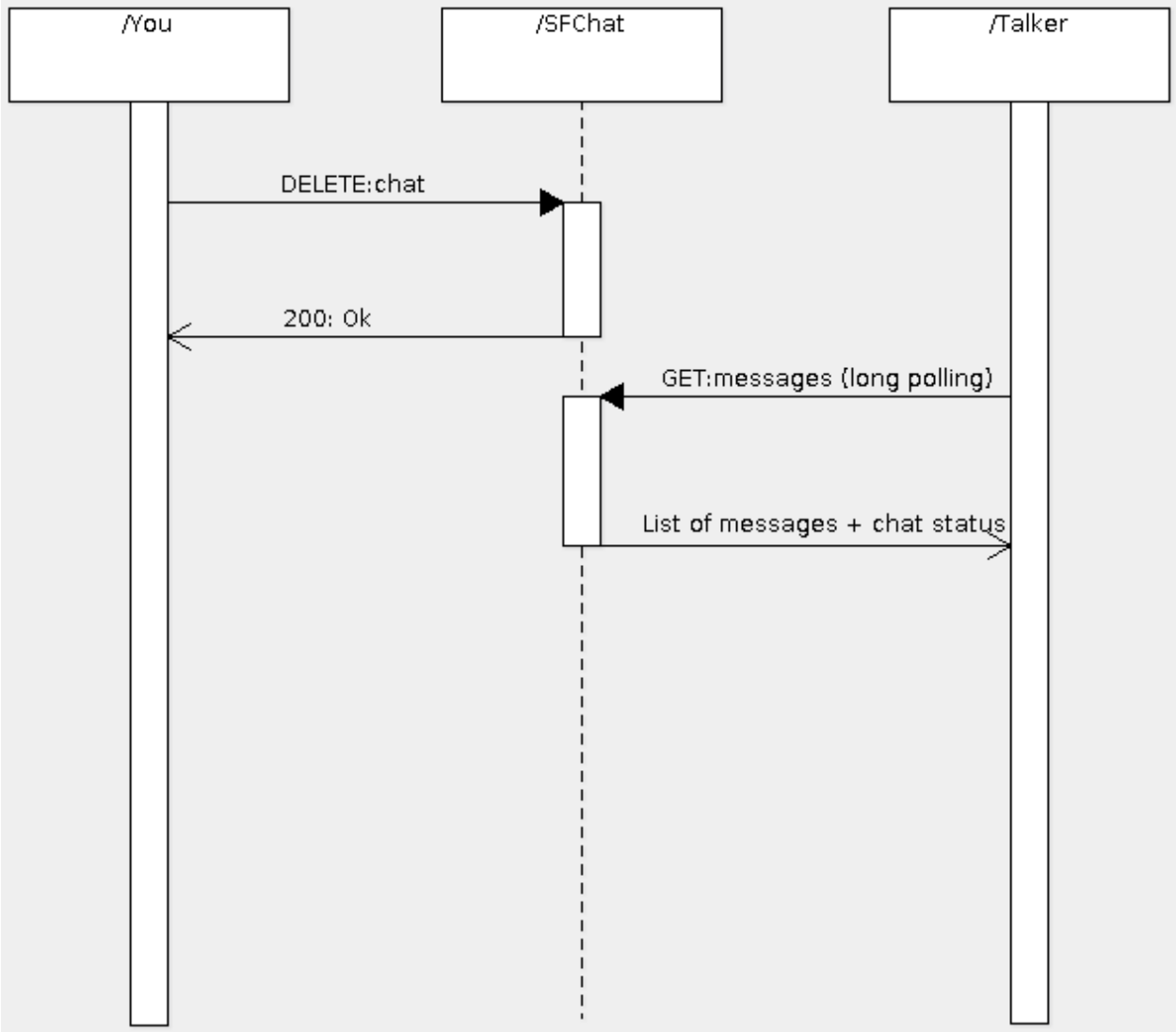
Sequence diagram authentication



Sequence diagram (conversation)



Sequence diagram (exit)



Database

Mongodb

Mongodb is a schema-less database. Therefore shema should be described in application. SFChat database has only one collection “chats” example below shows document in that collection:

```
{
        _id: ObjectId("507f191e810c19729de860ea"),
        status: “ready”,
        user_tokens: [
                ObjectId("507f191e810c19729de860eb"),
                ObjectId("507f1f77bcf86cd799439011")
```

```
        ],

        messages: [
                {
                    _id: ObjectId("507f191e810c19729de860ed"),
                    user_token: ObjectId("507f191e810c19729de860eb"),
                    msg: "Hi, how are you?",
                    system: false
                }
        ],
        long_polling: [
                {
                    _id: ObjectId("507f191e810c19729de860ed"),
                    user_token: ObjectId("507f191e810c19729de860eb"),
                    created: ISODate("2012-04-03T02:05:06Z")
                }
        ]
        created: ISODate("2012-04-03T02:05:06Z")
};
```

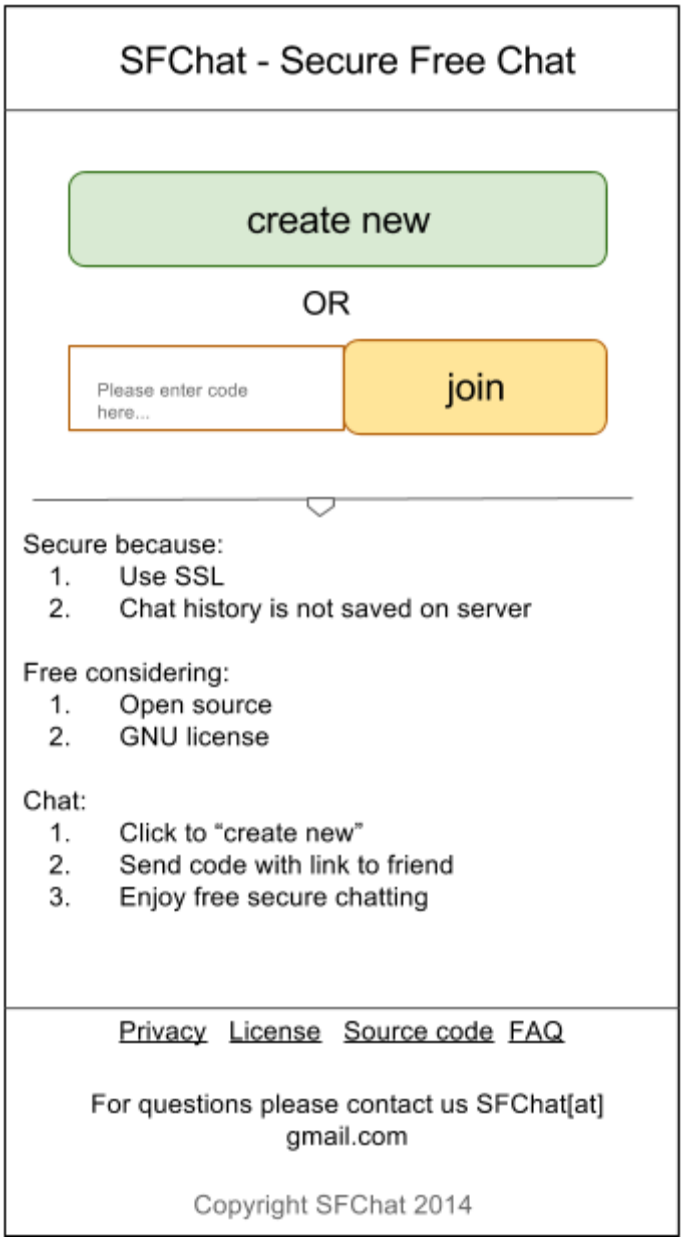
Table below displays schema for “chats” collection:

Name	Type	Description	Comment
chats._id	ObjectId	Unique key. Automatically generated by MongoDB	
chats.status	“draft”, “ready”, “closed”	Chat status. It should be one of the list	
chats.use_tokens	Array	List of “user tokens”. Generated by ObjectId().	It is used for authentication process.
chats.messages	Array	Messages container that include income messages	
chats.messages._id	ObjectId	“message token”. Generated by ObjectId().	
chats.messages.user_token	ObjectId	“user token”	For whom that message is

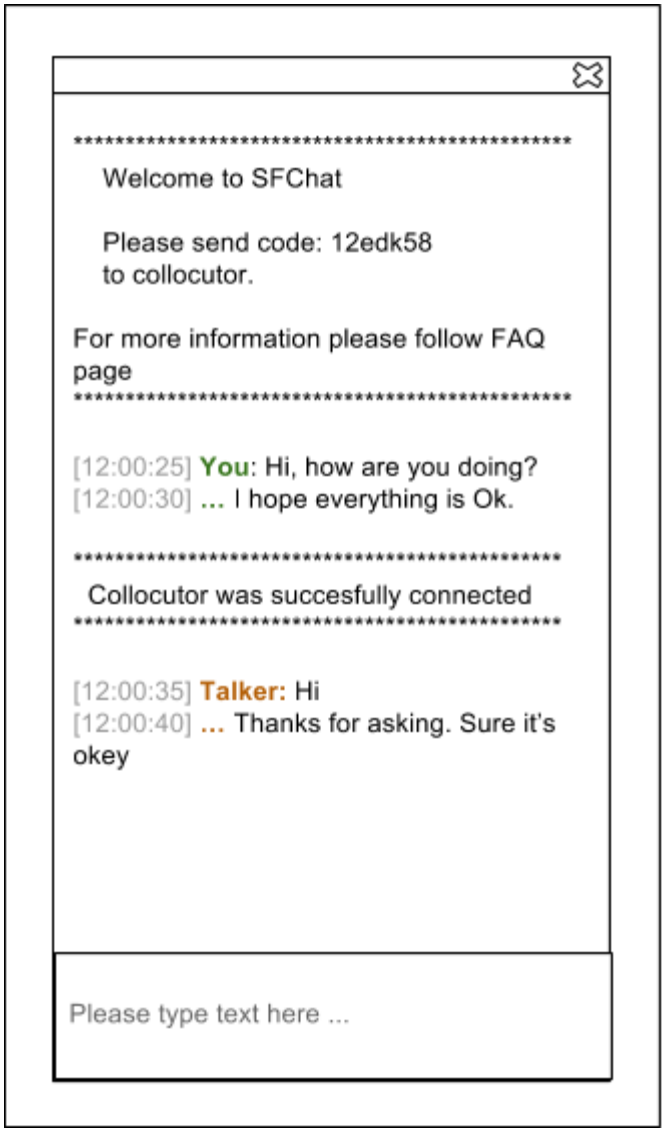
			addressed
chats.messages.msg	String[140]	User's message	
chats.messages.system	true, false	Indicate is that message system or not.	
long_polling	Array	Long polling process container	
long_polling._id	ObjectId	Long polling process identifier	
long_polling.user_token	ObjectId	"user token"	
long_polling.created	String[UTC]	Creation date in UTC format	Used to determine close tab or browser
chats.created	String[UTC]	Creation date in UTC format	It's used to identify old chats and clear them.

Mockups

Main Page



Chat page



Other pages

Others pages like “Privacy”, “License”,... etc are simply text page with header and footer. Such pages aren’t saved in Database they are html files. Because such pages are not dynamic they should be cashed.

MongoDB workflow

Such section display workflow in MongoDB point of view.
First off all tun mongo and use sfchat:

mongo
use sfchat

Create chat

To create chat we should generate “user token” and add them to chat:

```
var chat_token = ObjectId();
var user_token = ObjectId();
var message = {_id: ObjectId(), user_token: user_token, msg: "Welcome to SFChat <br /> Please send code: " + chat_token + " to Talker", system: true };

var chat = {
        _id: chat_token,
        status: "draft",
        user_tokens: [user_token],
        created: new Date()
};
chat.messages = [message];
db.chats.insert(chat);
```

To see that data has been successfully saved please run such command:

```
db.chats.find().forEach(printjson);
```

As a result we have “user token”, “invitation code” (chat token) and register system message:

```
{
        "_id" : ObjectId("543e33a2e3ce324d374246fc"),
        "status" : "draft",
        "user_tokens" : [
                ObjectId("543e33ace3ce324d374246fd")
        ],
        "created" : ISODate("2014-10-15T08:43:44.202Z"),

        "messages" : [{
                        "_id" : ObjectId("543e33b4e3ce324d374246fe"),
                        “user_token”: ObjectId("543e33ace3ce324d374246fd")
                        "msg" : "Welcome to SFChat <br /> Please send code: 543e33a2e3ce324d374246fc to Talker",
                        "system" : true
                }]
}
```

To get string value of ObjectId it is need read property “str”:

```
chat_token.str
```

Authorization

Each request to SFChat api contains “user token”, “chat token”. If, for instance SFchat api gets:

“chat token” = 543e33a2e3ce324d374246fc,
“user token” = *543e33ace3ce324d374246fd.*

Therefore it is need to run verification:

```
db.chats.findOne({
        $and: [
                {_id: ObjectId("543e33a2e3ce324d374246fc")},
                {status: {$in: [“draft”, “ready”]}},
                {user_tokens: ObjectId("543e33ace3ce324d374246fd")}
        ]},
        {_id: 0, messages: 1}
);
```

As a result if null that authorization data is invalid otherwise return list of messages.

Join chat by code

To join chat it is need to verify:

- “invitation code”
- number of users in chat

then generate user token and registry system message that chat is ready.

Verification:

```
db.chats.findOne({
        $and: [
                {_id: ObjectId("543e33a2e3ce324d374246fc")},
                {status: “draft”}
        ]},
        {_id: 1}
);
```

If result is null then “invitation code” is invalid otherwise it should be checked number if user_tokens. Such number should be 1.

Next generate “user token”, update chat status and register system message for “chat creator”. To add new message it should be used “$push” operator:

```
var user_token_joiner = ObjectId();
var message_ready = {
        "_id" : ObjectId(),
        “user_token”:  user_token_joiner,
```

```
        "msg" : "Talker was successfully joined to chat",
        "system" : true
};

db.chats.update(
        {_id: ObjectId("543e33a2e3ce324d374246fc")},
        {
                $set: {status: "ready"},
                $push: {user_tokens: user_token_joiner},
                $push: {"messages": message_ready}
        }
);
```

Conversation long-polling

Conversation is based on long polling. Object “long-polling” in the “chat” collection contains attive long polling processes. It is essential to prevent runs several long polling after refresh chat page.

Table below describes key points of long polling:

Event	FrontEnd	BackEnd
Start process	1. User on a chat page 2. Ajax request runs to server to gets messages 3. Then runs Ajax to delete messages 4. Finally runs get messages again	1. Removes all registered long polling processes for current users 2. Start new long polling process 3. Long polling accordingly two configuration parameters: number of iterations and sleeping time (for more information please look into corresponding section)
End process	1. Ajax response has 403 code 2. Chat has “close” status	1. Chat was closed 2. Tab of browser close was detected

Note: close tab or browser cause close chat only if one of the chat participant has still opened chat. In the case when all chat’s participants have closed chat SFChat uses Python-RQ task to clear garbage data.

Django configuration

Configuration is in base.py file:

```
SFCHAT_API = {
        'authentication': {
                'user_token_header': 'HTTP_X_SFC_USERTOKEN',
                'chat_token_header': ‘HTTP_X_SFC_CHATTOKEN’
```

```
        },
        'long_polling': {
                'sleep': 3,
                'iteration': 60
        },
}
```

Configuration description:

Key	Type	Description
authentication.user_token_header	Sting[32]	Header name for “user token”
authentication.*chat_token_header'*	String[32]	Header name for “chat token”
long_polling.sleep	Integer, sec	Number of second that indicates how long long polling iteration should sleep before start new one
long_polling.iteration	Integer	Number of long polling iterations for one process

Note: multiplies long_polling.sleep with long_polling.iteration show how long long polling process runs. Time that necessary to figure close tab and browser is calculated as:

*2 * long_polling.sleep * long_polling.iteration*

FrontEnd

All frontEnd components should be:

1. supported cross-browsers last two version of each populars ones (Chrome, FF, Opera, Safary, IE)
2. supported different desktop and mobile OS
3. minified
4. gziped
5. applied Content Security Policy

CSS HTML

All CSS should follow Google recommendation.

JScript

JScript should follow as much as possible manifesto.

Each new JS class should be property of SFChat object. That object is a “namespace”. So /js/sfChat.js:

```
var sfChat = {};
```

Others modules in that case for instance /js/sfChat/validator.js:

```
sfChat.validator = {
        ….
};
```

Google Analytics

To know what it's going on on a page without damage privacy. Google analytics (GA) will not be used for chat page. Moreover GA will collect events:

- Create new chat
- Join to the chat

Task Queue

Tasks that should run on background with configure scheduler. For monitoring Queues it is used Python-RQ. Removes all chats, which are status "closed". And that is not closed but there are more than 24 hours. Please look into resources section for more links.

Chat termination

Remove all information about chat if it was created 1 day ago. This limitation also should be on the page with politics.
Workflow of chat termination has 2 phases:

- Remove messages and set chat to status "closed"
- Register "System message" that chat was automatically closed
- Remove all chats that do not have any messages

So the first run of task removes all undelivered messages. It helps clear chat history in FrontEnd. The second run completely remove information about chat.

Installation

1. Install dependencies: sudo pip3 install -r config/requirements.pip
2. Bower
    a. sudo apt-get install nodejs
    b. sudo apt-get install npm
    c. sudo npm install bower -g
    d. sudo ln -s /usr/bin/nodejs /usr/bin/node
3. Use bower to install JQuery: cd ./bin/bower && bower install
4. Install Python-RQ:
    a. run /bin/ install-redis-server-local.sh to install Redis locally
    b. run $ *sudo pip3 install django-rq*
5. Usage Python-RQ:
    a. run redis-server with command $ *~/redis-2.6.16/src/redis-server*
    b. run local django-server with $ *make*
    c. running workers in another terminal $ *python3 manage.py rqworker default*