# Chat settings

Scope: this doc covers permissions and configuration that is specific for one contact or group, and should or should not be taken into account when sending messages.

## Problem

Certain chat features are not desirable for some contacts/groups, some other require contact specific configuration.

These settings can be symmetric or asymmetric. Symmetric settings require mutual agreement. Asymmetric can be set by one party independently of the other. The focus of this RFC is asymmetric settings, as they are simpler - they require one-way notification for their change.

These settings can be local or remote. Local settings only affect chat functionality for a given contact or group, but are not known to the remote parties. For each type of local setting user only needs one value per chat. Remote settings should be taken into account when sending the message, to avoid rejected/ignored messages. For each type of remote setting user needs two values per chat - the value received from the other party (it should be taken into account when sending the message) and the te value sent to other party (that should be taken into account when processing received messages).

Local settings can be implemented as an extension of global user settings, they are not the focus of this RFC.

This RFC focus is asymmetric remote settings that include:

- permission to send voice messages - some users prefer to not receive them.
- permission to delete sent messages and the maximum duration when it is allowed.
- permission to send images - e.g. some automatic clients/bots may not support images, and this would explicitly prohibit it.
- permission to edit sent messages and the maximum duration when it is allowed.

These permissions are not taken into account for group memberships, instead group permissions are that are set by group owners.

## Solution

### Protocol

Broadcast user settings and preferences in the same way as profile updates by adding `preferences` property alongside `profile` property - it will be sent as part of `x.info` message.

For groups these are also added to group profile and sent via `x.grp.info` message.

`preferences` property is a dictionary with boolean or number values - clients must ignore unknown values.

Current schema for `preferences` member:

```
js { "definitions": { "enabled": { "properties": { "enable": {
"enum": ["on", "off"] } }, "additionalProperties": true } },
"optionalProperties": { "voice": { "ref": "enabled" } // "image":
{ "ref": "enabled" }, // "file": { "ref": "enabled" }, //
"delete": { "ref": "enabled" }, // "acceptDelete": { "ref":
"enabled" }, // "edit": { "ref": "enabled" }, // "receipts": {
"ref": "enabled" } }, "additionalProperties": true }
```

`enable` enum can be potentially extended to mutual

Every time user updates the settings and update profile should be sent to affected contacts.

### Database schema

```sql
ALTER TABLE users ADD COLUMN preferences TEXT;

ALTER TABLE group ADD COLUMN preferences TEXT;

ALTER TABLE contacts ADD COLUMN userpreferences TEXT; ALTER
TABLE contacts ADD COLUMN contactpreferences TEXT;
```

Preferences in the user record would have all fields that the user supports, sent preferences on the contact level (`user_preferences`) - only those that are different from global. `contact_preferences` in contact will have all supported fields.

Group preferences are set by owners and do not depend on user preferences.

### Types

```haskell
data ChatPreferences = ChatPreferences { voice :: Maybe
Preference -- only this field is needed right now -- image :: Maybe
Preference, -- file :: Maybe Preference, -- delete :: Maybe Preference, --
```

acceptDelete :: Maybe Preference, -- edit :: Maybe Preference, -- receipts :: Maybe Preference }

data Preference = Preference {enable :: PrefSwitch}

data PrefSwitch = PSOn | PSOff -- for example it can be extended to include PSMutual, that is only enabled if it's enabled by another party ```

## UI

UI in the contact will need to differentiate between on/off and unset values in which case the global setting will be used.

## API

`/_set prefs <json> /_set prefs @1 <json> /_set prefs #1 <json>`

Optionally, it may help to have this type to send to the UI merged preferences:

```haskell data MergedChatPreferences = MergedChatPreferences { voice :: MergedPreference }

data MergedPreference = MergedPreference { value :: Preference, global :: Bool } ```

But it may be easier to merge in place in the UI from two ChatPreferences values.
```