Messaging Microservice

Interface Identity

The **messaging interface** is the main interface that controls the messaging, announcements, and emergency alert features of the website. This interface includes all endpoints for users subscribing and unsubscribing to channels, sending messages, reviewing history, and checking online presence.

Resources

There are 5 different resources that are part of the messaging interface:

- publish(channel: string, message: Message): Promise<void>
- subscribe(channel: string, userId:string, onMessage: (message: Message)=> void): Promise<Subscription>
- unsubscribe(subscription: Subscription): Promise<void>;
- history(channel: string, limit?: number): Promise<Message[]>
- presence(channel: string): Promise<PresenceData[]>

The following provides more details about each of the resources in the messaging interface.

publish

Syntax

```
publish(channel: string, message: Message): Promise<void>
```

Semantics

The publish operation broadcasts a message to a specific channel
Parameters:

- *channel*(String): The name of the channel
- *message*(Message) The message to be sent

Preconditions:

- The *channel* parameter must be an existing channel, and the message must be a valid format

Postconditions:

- Returns a Promise with the status of the publish

Error Handling:

- 400 Bad Request: If the *channel* or *message* parameter are not valid, an error is returned
- subscribe(channel: string, userId:string, onMessage: (message: Message)=> void): Promise<Subscription>

subscribe

Syntax

```
subscribe(channel: string, onMessage: (message: Message)=> void ):
Promise<Subscription>
```

Semantics

The subscribe operation subscribes a user to a channel

Parameters:

- *Channel* (String): Name of channel
- *onMessage:* (message: Message)=> void) function to handle messages being sent

Preconditions:

- The *userId* is a valid user, the *channel* exists, and the onMessage function is valid

Postconditions:

- Returns a *Subscription* containing details about the channel and user

Error Handling:

- 400 Bad Request: If any of the parameters are not valid, an error is returned

unSubscribe

Syntax

```
unSubscribe(subscription: Subscription):  Promise<void>
```

Semantics

The *unSubscribe* operation unsubscribes a user from a channel.

Parameters:

- *Subscription* (Subscription)

Preconditions:

- The *subscription* parameter must be an existing subscription

Postconditions:

- Returns a *Promise* containing the status of the operation

Error Handling:

- 404 Not Found: If the *subscription* parameter doesn’t exist in the database

history

Syntax

```
history(channel: string, limit?: number): Promise<Message[]>
```

Semantics

The history operation adds returns messages broadcast in the channel.

Parameters:

- *Channel* (String): channel to retrieve history from
- *Limit* (Number): max amount of messages to return

Preconditions:

- The *channel* parameter must be a valid channel and *limit* if included must be a valid number

Postconditions:

- Returns message history

Error Handling:

- 400 Bad Request: The *channel* parameter is not a valid channel

presence

Syntax

```
presence(channel: string): Promise<PresenceData[]>
```

Semantics

The presence operation detects which users are online in a specific channel

Parameters:

- *Channel* (String): channel in which the presence is being checked

Preconditions:

- The *channel* parameter is an existing channel

Postconditions:

- Returns data on who is online

Error Handling:

- 400 Bad Request: The *channel* parameter is not a valid channel

Data types and constants

The messaging interface depends on the Message, Subscription, and PresenceData interfaces which define the different data structures used in the messaging system. The interface are defined below:

```
interface Message {
  name?: string;
  data: string;
  id: string;
  clientId?: string;
  connectionId?: string;
  timestamp: number;
  extras?: Record<string, unknown>;
  encoding?: string;
}
```

Each of the parameters part of the Message interface serves a specific purpose in the messaging system:

- name: This is an optional field representing the name of the message.
- data: This is the contents or payload of the message.
- id: This is a unique ID assigned by Ably to each message.
- clientId: This is an optional field representing the ID of the client that published the message.
- connectionId: This is an optional field representing the ID of the connection used to publish the message.
- timestamp: This represents the timestamp when the message was received by Ably, in milliseconds since the Unix epoch.
- extras: This is an optional field containing a JSON object of arbitrary key-value pairs, which might include metadata.
- encoding: This is an optional field indicating encoding information. Typically, it is empty since messages are automatically decoded by Ably.

```
interface Subscription {
  channel: string;
  unsubscribe: () => Promise<void>;
}
```

The Subscription interface provides details about a particular subscription:

- channel: This represents the name of the channel to which the subscription belongs.
- unsubscribe: This is a method that allows unsubscribing from the channel.

```
interface PresenceData {
  clientId: string;
  status: 'online' | 'offline';
  timestamp: Date;
}
```

The PresenceData interface provides information about the presence of clients in a channel:

- clientId: This represents the ID of the client.
- status: This indicates the status of the client, which can be either 'online' or 'offline'.
- timestamp: This provides the timestamp of the last status update.

Error handling

There are two main errors that can be raised from multiple resources on the interface, a 400 Bad Request and a 404 not found

If an incorrect or unrecognized data type is passed through any of the resources, a 400 Bad Request error will be returned.

A 404 Not Found Error occurs when the formatting is valid but the requested object doesn’t exist in the database

Variability

The Messaging interface introduces very little variability, with the only optional parameter being the number of messages shown when fetching conversation history

Quality attribute characteristics

Performance

The messaging interface aims to provide unnoticeably small delivery times for sending and receiving a message, and minimal wait time when loading and starting conversations.

Publishing a message and checking online presence should complete within 0.1 seconds, Subscribing and Unsubscribing should be completed within 1 second, and history for conversations should take at most 5 seconds to complete, with shorter times expected for smaller conversations.

Availability

Users should always be able to rely on the emergency alert feature to send an alert, so the messaging microservice aims for a 99.99% uptime. This is supported by Ably's commitment to 99.999% uptime and vercel's commitment to 99.99% uptime.

Rationale and design issues

The messaging interface follows a standard subscription-publisher data flow, which was chosen due to its versatile use cases. 1:1 messaging, emergency alerts, and application wide announcements can all be implemented using the subscription model which allows for reuse of code. The data types were defined with anything not absolutely necessary as optional to allow for a simple integration experience.

The interface was also defined in a simple and modular way, with many different features being implemented using only 5 core functions: publish, subscribe, unsubscribe, presence, and history. This allows for a much simpler onboarding process for new developers, and allows for good maintainability.

Usage guide

Here are some examples of how to use some of the resources in the messaging interface written in TypeScript.

publish

```
const chatMessage: Message = {
    data: 'Hello, world!',
    id: 'abcd1234',
    timestamp: Date.now()
};

messagingService.publish('general-chat', chatMessage).then(() => {});
```

subscribe

```
messagingService.subscribe('general-chat', (message: Message) => {
    console.log(`Received message: ${message.data}`);
}).then(subscription => {
    // Handle subscription
});
```

unSubscribe

```
messagingService.unsubscribe(subscription).then(() => {
    console.log('Unsubscribed successfully');
});
```

history

```
messagingService.history('general-chat', 10).then(messages => {
    messages.forEach(message => {
        console.log(`Message ID: ${message.id}, Data: ${message.data}`);
    });
});
```

presence

```
messagingService.presence('general-chat').then(presenceDataArray => {
    presenceDataArray.forEach(data => {
        console.log(`Client ID: ${data.clientId} is currently 
${data.status}`);
    });
});
```

modifyEvent

```
const oldLunchEvent: Event = {
    eventId: '1234',
    eventName: 'Breakfast: poached eggs and toast',
    startDateTime: '2023-10-09T09:00:00',
    endDateTime: '2023-10-09T11:00:00',
    isMealEvent: true
};

const newLunchEvent: Event = {
    eventId: '1234',
    eventName: 'Breakfast: granola and fruit',
    startDateTime: '2023-10-09T09:00:00',
    endDateTime: '2023-10-09T11:00:00',
    isMealEvent: true
};

eventsService.modifyEvent(oldLunchEvent, newLunchEvent).then(() => {});
```