

Chat Babel AWS Project

Jaime Gimillo Bonaque, Jakub Fołtyn, Kacper Grzymkowski

General Description

Serverless application that allows users to communicate by text, across languages using a machine learning translation model to translate messages. Users are able to create an account, and then log in. After logging in, they will be able to join other users in existing chat rooms, or even create their own. There they can send public messages that may be viewed by other logged in users in that room. What is more important, all users will have the possibility to translate the whole chatroom into their preferred language with a click of a button.

The system should support serverless architecture. To achieve this, most of the system backend will utilize AWS Lambda services to support all the message and user related actions (these include sending a message, creating a chat room, joining a chat room etc.). To further support the serverless aspect of our system, as well as to achieve higher availability and scalability, we have decided to use DeepL (Amazon Translate is unfortunately not available in the sandbox environment, otherwise it would be the more logical choice) service for translation purposes. This way we will not have to rely on our own deployed model and instead will be able to take advantage of ready-to-use services.

We will use Amazon Cognito for all the authentication purposes. In terms of storage (messages for translation, user data) DynamoDB Storage will be utilized.

Functional Requirements:

- Users of the application can create accounts and log in.
- Users can create, join and change chat rooms.
- Users can send messages and receive messages from other users in real time.
- Upon joining a room, the application displays the most recent messages that were sent before the join.
- Users can translate messages sent by other users to their preferred language by pressing a button.

Non-functional Requirements:

- Joining a room and loading previously sent messages in a room does not take more than 1 second, 99% of the time, 95% of messages are delivered within 1.5 seconds.
 - The time limits here were increased due to issues with the environment forcing the use of a less efficient architecture.
- Internationalization of the user interface text and proof-of-concept localization (using machine translation).
- Weekly backups of all persistent data set up for disaster recovery.

- Repeat translations (i.e. requested for the same language by two different users) do not call the translation API multiple times.
- Translation functionality can be tuned to tweak formality levels on a room by room basis.
 - Originally this requirement also covered profanity settings however due to environment issues we had to use a different translation service.

Technologies Planned:

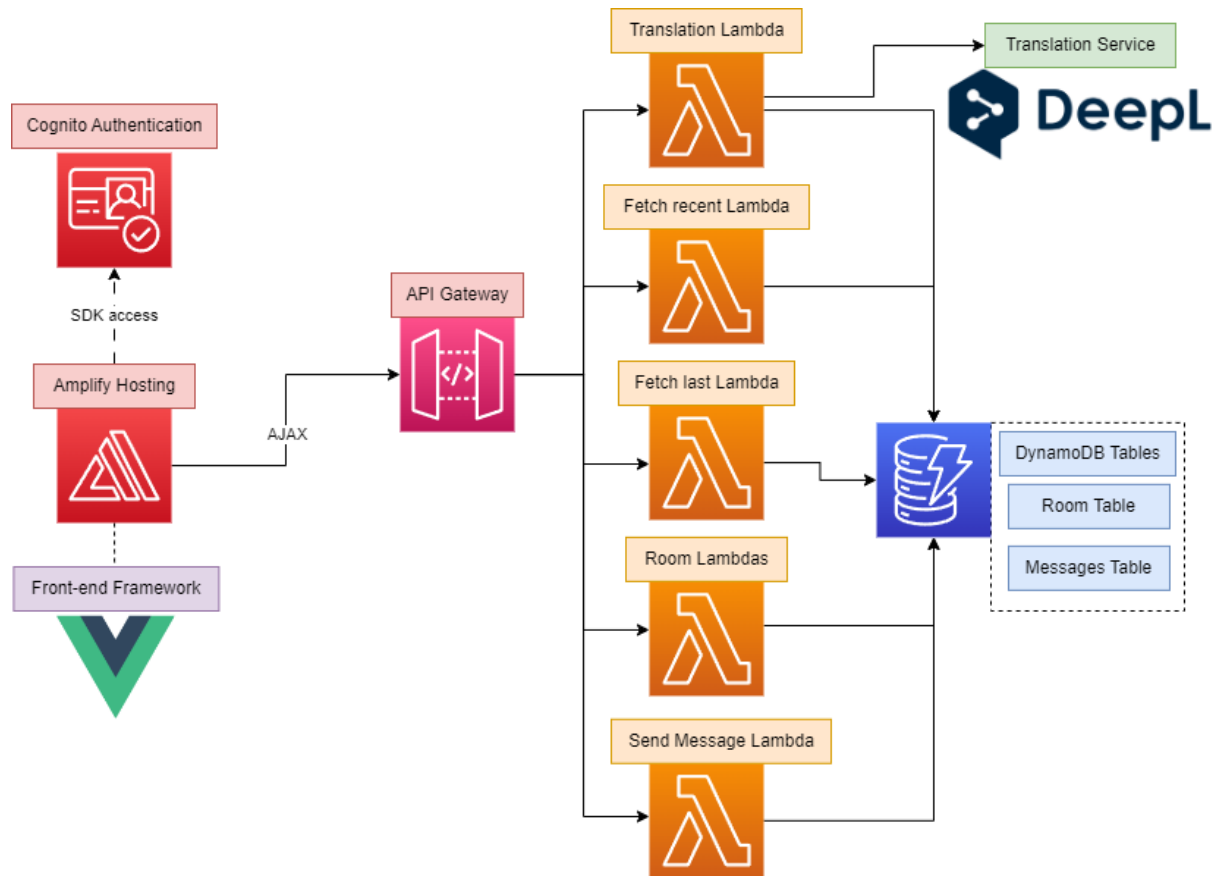
- Lambda backend
 - Using Chalice framework
 - Websocket (experimental support in the framework)
- Amazon Translate
- AWS CloudFormation (for some of the deployment)
- DynamoDB Storage
- Amazon Cognito

Technologies Used:

- Lambda simple REST backend
- DeepL Translate
- DynamoDB Storage
- Amazon Cognito
- Amazon Amplify Hosting

Architecture used

We decided to use a serverless architecture by taking advantage of multiple existing services, mostly focused on AWS, but we were forced to use an external service for translation. We used Vue as the front-end framework to create a statically served client. This client application was hosted using AWS Amplify Hosting. The client communicates with the backend via AJAX requests with the REST APIs exposed via the AWS API Gateway. The backend functionality is implemented using multiple AWS Lambda functions performing one single task each. DynamoDB is used for persistence, split between two tables - one for messages and one for rooms. The translation lambda then can invoke the appropriate DeepL API for the translation functionality. We implement asynchronous delivery of messages by having the client application send “refresh” requests on a short timer, which return recently sent messages.



Front-end details

As we mentioned before, we have used AWS Amplify for the hosting and continuous deployment of our web application. We initially planned to use this tool for designing the backend too, but it required creating additional roles (which is not allowed in the sandbox environment), so we finally had to configure each part from the backend separately and then integrate them. We couldn't either create a custom domain with Amazon Route 53, so the default domain provided by AWS Amplify is <https://main.dj55csosw9usx.amplifyapp.com/>. Our source code is located in a private GitHub repository, connected to AWS Amplify by the main branch, so that it rebuilds the application with every new commit.

The web framework chosen has been Vue, a well-known Javascript framework based on rendering and component composition, and for styling our components more easily we have used Bootstrap. The structure of the URL paths and corresponding components is the following:

- '/' - Home.vue (welcome and registration)
- '/login' - Login.vue
- '/rooms' - Room.vue (list of rooms and creation form)
- '/rooms/:roomName' - Chat.vue

Our first challenge was supporting registration of new users and their subsequent login. We managed to meet this requirement with the help of Amazon Cognito. However, since we couldn't use the backend tools from AWS Amplify, we had to communicate with the users pool with a Javascript SDK. The implementation can be found in the file `cognito.js` and the functions `login()` and `register()`, in the components `Login.vue` and `Home.vue` respectively. We configured the pool in such a way that:

- Users must specify username (unique in the pool), email address and password to register.
- Before logging in, recently registered users must verify their account by clicking a link sent to their emails.
- Users can login with both username and email (and password obviously).
- All fields from the forms are validated on submit, displaying information messages to the user in case of error and success.

In order to make the credentials of a logged in user persistent (after a refresh they are not lost), we have used the local storage of the browser, remembering to delete them once the user logs out. When the main component (`App.vue`) is mounted, we check in the browser local storage if there exists some credential. The same logic has been applied to the language preference, setting it to English by default. It must be remarked that once the user logs in, the access token is actually not used for anything other than verifying that it exists. In other words, creating an account in our system is only needed to actually start using the application, even though the API calls can be made by anyone (no credentials needed).

Jumping right into the code, the file `auto_localize_deepl.py` allowed us to provide a `locations_source.json` (UI fields written in English) to the DeepL Python library so that it generates the locations for the closed set of languages we support (English, Polish, Spanish and French). This way, all the components can access the text fields in the preferred language by accessing the `locations.json`.

As for the creation and list of rooms (in component `Room.vue`), the most important functions are:

- `fetch_rooms()` - mainly calling a GET to the path `'/getRooms'`. It is only triggered when reloading the component.
- `create_room()` - basically making a POST to the path `'/createRoom'` when the creation form is submitted. Again, any error on the creation form is managed so that the user gets an informative response, and if the creation was successful it redirects the user to the new chat room.

Regarding the chat room, one of the most important methods in `Chat.vue` is `fetch_messages()`. As its name suggests, it calls the GET method to the path `'/getMessages?RoomID=<uuid:roomId>'`, getting the most recent messages from the chat. This action is performed every half second by setting a timeout when the component is mounted. The other main function is `translate_message()`, requesting for the translation of a specific message and updating its translation field with the returning value.

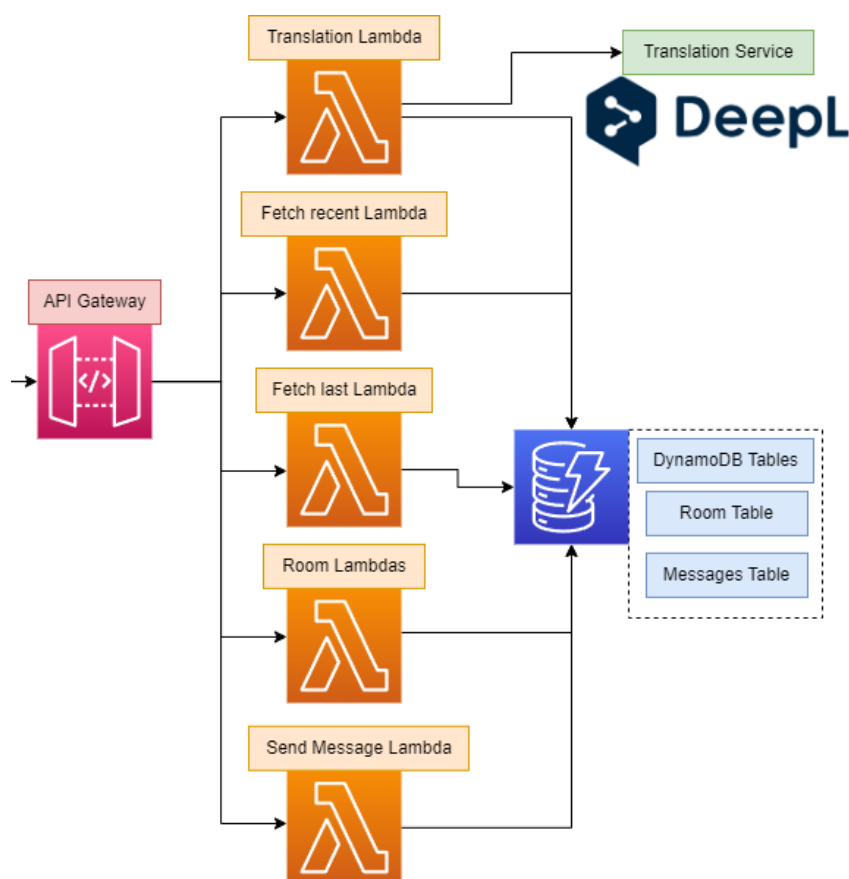
There is an implementation detail with these two functions that is not as elegant as we wish, but we haven't come up with any better solution if we wanted to preserve the translating loader (spinner that shows up while the translation is taking place). Since during the

translation of a message (which might take some time) we will probably fetch the room messages, the attribute 'translating' from the message (which shows or hides the translating loader) might be wrongly modified. That's why we decided to use a simple mutex, which doesn't allow previous functions to modify the messages attribute from the component at the same time.

To conclude, just mention some of most important modules that have been used are:

- vue-router: for linking the routes paths to the corresponding views and redirecting if necessary (wrong URL or no permission).
- vuex: a really useful state management library, used in our case by all components to access the credentials of the user (if logged in) and the current preferred language.
- axios: HTTP Client for sending the requests to the REST API.
- vue-spinner: library for the loading spinners.

Back-end details



Backend, as presented on the schema above, consists of *DynamoDB*, *API Gateway* and multiple *AWS Lambda functions*. *DynamoDB* serves as primary storage for both all the rooms created by users, as well as all the messages sent. *Lambda functions* are used for all the different operations involving data processing and communication with the database, such as sending new messages, retrieving all messages and message translation. Finally,

API Gateway serves as an entry point to access all the *Lambda functions*. Let us now discuss each element in greater detail.

DynamoDB

ROOMS	
NAME	<str> room name, unique
Formality	<bool> formality setting

As previously stated, there are two tables in our database. The first table, ROOMS (table above), stores data about rooms created by users. Each row stores room name, which is unique and is considered to be its id, and Formality, which is a boolean flag which indicates whether a room has formality setting turned on or not (it affects translations).

MESSAGES	
ID	<str> message ID, unique
ROOM_ID	<str> room name (in which message was send)
Sender	<str> sender name
Text	<str> message text
Time	<int> message sending time (in seconds)
Translations	<list> message translations (en, fr, es, pl)

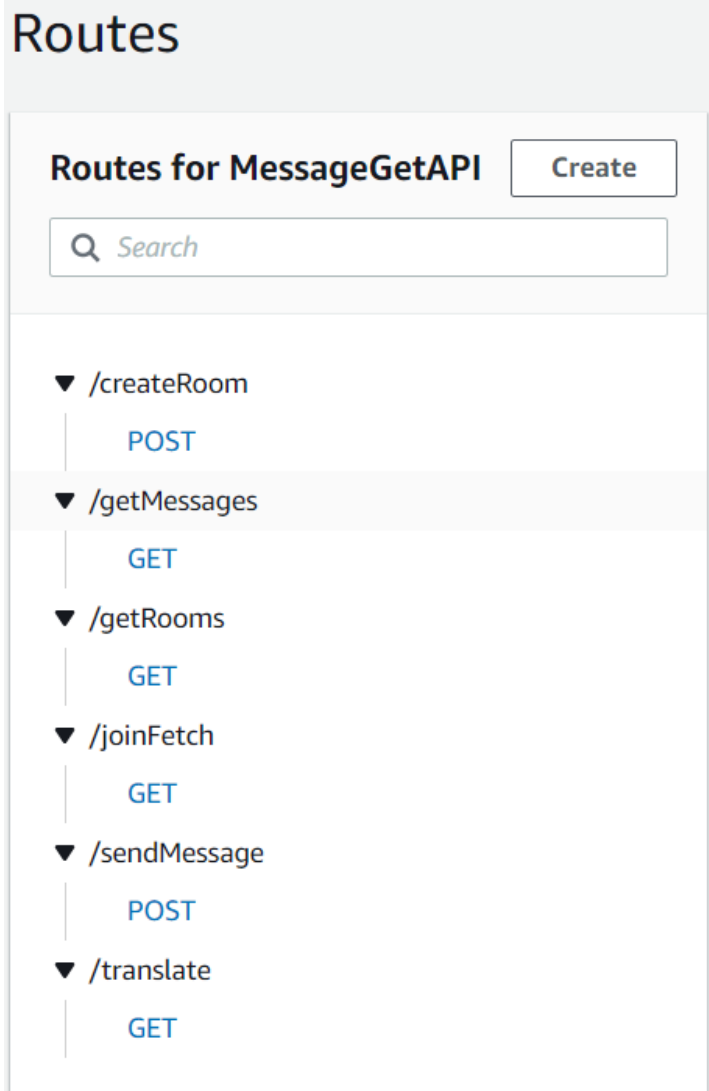
The second table, MESSAGES (table above), stores the messages sent by users. Each row consists of message ID, room id in which the message was sent (the same as room name), name of the sender (user) of the message, text of the message, time it was sent (in seconds) and finally a list of available translations. Those translations are stored in order not to call the translation API for already translated messages. Available languages are: Spanish, English, French and Polish. Message id and time sent are obtained via request parameters to the *Lambda function* adding the message to the database.

It is also worth noting that an additional index was created for this table, enabling us to sort its contents by the message time. This way, we are able to return n last messages in a room.

API Gateway

We used the *AWS HTTP API Gateway* to create API endpoints to connect to our *Lambda functions*. This gateway remains publicly available, mostly for testing purposes. We created multiple routes, as presented on the image below: each route corresponds to a separate

Lambda function and uses a different method. The URL address for our gateway is: <https://ek5ajs509b.execute-api.us-east-1.amazonaws.com>, and to connect to different function endpoint one must add /<route_name> at the end. It is important to note, however, that this is not the intended way of using our application, as the front-end components are better suited for communication with our API and should be used instead.



Lambda functions

Lambda functions are the main backend component, as they are responsible for communicating with and managing the database. There are 6 functions in total, and they will be described in more detail in dedicated subsections. All *Lambda functions* were built using Python 3.10, we also used requests package for API calls and built-in (in lambda environment) *boto3* package for communication with *DynamoDB*. For json validation we used the *jsonschema* package.

Get latest messages

Function used to get the latest messages when first joining a room. It uses the GET method and takes the room name as a request parameter. Utilizing the time index set on the MESSAGES table, it returns the latest 25 messages in the room.

Name	fetch_latest_messages
URL	https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/joinFetch
Method	GET
Params	<ul style="list-style-type: none">• RoomID – room name from which to take messages
Returns	<ul style="list-style-type: none">• List of messages
Functionality	<ul style="list-style-type: none">• Scan the MESSAGES table by the time index.• Return the latest 25 messages sent within the requested room.

Get recent messages

Function used to get the most recent messages (from the last hour). Frontend components invoke it every 0.5 seconds to get the most recent messages in a room to the user. It uses the GET method and takes room name and the number of seconds as parameters. By the number of seconds we mean the time within which the message needed to be sent, in order to be returned by this function (so a value of 200 means that messages from the last 200 seconds will be returned). It defaults to 3600 seconds (one hour).

Name	get_room_messages
URL	https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/getMessages
Method	GET
Params	<ul style="list-style-type: none">• RoomID – room name from which to take messages• Seconds – time messages sent within which are to be returned, maximum is 3600 seconds (so messages sent up to 3600 seconds ago are returned)
Returns	<ul style="list-style-type: none">• List of messages
Functionality	<ul style="list-style-type: none">• If invalid number of seconds was passed (or even no value was passed) assume number of seconds to be 3600• Scan MESSAGES table for messages with matching room ID and sent time within the time limit• Return found items as list

Send message

Function used to send a new message. Sending a new message means putting a new record in the MESSAGES table. It uses the POST method and takes room name, sender name and message text as parameters. Those parameters are to be passed as a json file in the request body. The function then adds some additional parameters to the message and puts it in the MESSAGES table.

Name	send_message
URL	https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/sendMessage
Method	POST
Params	<ul style="list-style-type: none">• RoomID – room name in which the message was sent• Sender – name of the sender of the message• Text – message text
Returns	<ul style="list-style-type: none">• Sent message ID
Functionality	<ul style="list-style-type: none">• Validate passed JSON schema• Add time info (request time epoch)• Add ID (request ID)• Add translations (a list of Nulls)• Put created JSON into MESSAGES table

Translate message

Function used to translate messages. It uses the GET method and takes message id and target language code as parameters. It then checks if a translation already exists for the given message (and returns it if it does). If it does not, then it accesses the formality settings in the message room and passes it (as well as message text) to the DeepL API request. The response with the translation is then updated to the database and returned.

Name	translate_text
URL	https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/translate
Method	GET
Params	<ul style="list-style-type: none">• ID – id of message to be translated• lang – language to which the message is to be translated (<i>English, French, Spanish or Polish</i>)

Returns	<ul style="list-style-type: none"> Translated text
Functionality	<ul style="list-style-type: none"> Check if message with that ID already has translation in that language (if it does, return translation) Access room formality settings Access secret API key (from SecretsManager) Invoke DeepL translate API (get translation as response) Update translation in MESSAGES table

Create room

Function used to create a new room. It uses the POST method, and takes room name and formality settings as parameters (json in request body). It first checks if a room already exists (by checking passed name), and if it does not, creates it (by putting new record in the ROOMS table).

Name	create_room
URL	https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/createRoom
Method	POST
Params	<ul style="list-style-type: none"> NAME – room name Formality – formality setting
Returns	<ul style="list-style-type: none"> Message that a room was created/fail message
Functionality	<ul style="list-style-type: none"> Passed JSON is validated Check if room with passed name exists (if it does, error is returned) Add room to ROOMS table

Get room list

Function used to retrieve the list of available rooms. It uses the GET method, and does not take any parameters. It simply scans the ROOMS table and returns it in its entirety.

Name	get_rooms
URL	https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/getRooms
Method	GET

Params	<ul style="list-style-type: none"> • <empty>
Returns	<ul style="list-style-type: none"> • List of rooms
Functionality	<ul style="list-style-type: none"> • Scans the ROOMS table and retrieves it in its entirety

Tests

Lambda functions tests

Let us first conduct some *Lambda functions* tests, to assure that all functions work as intended. Those tests were conducted by making a request to the API endpoint corresponding to each *Lambda* with *Postman*, and then checking the results.

Create room

First we invoke the Create Room function, and create a new room with the name “lambdatests”. As we can see, the message appears that a room has been added, and indeed when checking *DynamoDB* we can see our new room.

The image shows a Postman API client interface at the top and the AWS DynamoDB console at the bottom.

Postman Interface:

- Method:** POST
- URL:** `https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/createRoom`
- Body (Text):**

```

1 {
2   "NAME": "lambdatests",
3   "Formality": false
4 }

```
- Status:** 200 OK, Time: 1881 ms
- Test Results:**

```

1 Room lambdatests has been added

```

DynamoDB Console:

- Table:** Rooms
- Items returned (11):**

NAME	Formality	Profanity
abc	false	false
asdadasda	false	false
Example	false	true
Example2	false	true
NewRoom	true	
testyopwji	false	
AAAAAAAAA	false	
PerfTest	false	

Get room list

We invoke the *Get Rooms* function, and as we can see, our newly added room appears on the returned list.

GET ▼ <https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/getRooms> Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cooki

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (5) Test Results Status: 200 OK Time: 960 ms Size: 753 B Save as Example

Pretty Raw Preview Visualize Text ▼ 🔍 🗑

```
1 [{"items": [{"Formality": false, "NAME": "zleparametry"}, {"Profanity": false, "Formality": false, "NAME": "abc"}, {"Profanity": false, "Formality": false, "NAME": "asdasdasda"}, {"Profanity": true, "Formality": false, "NAME": "Example"}, {"Profanity": true, "Formality": false, "NAME": "Example2"}, {"Formality": true, "NAME": "NewRoom"}, {"Formality": false, "NAME": "testyyopwji"}, {"Formality": false, "NAME": "AAAAAAAAAAAA"}, {"Formality": false, "NAME": "PerfTest"}, {"Profanity": false, "Formality": true, "NAME": "Example3"}, {"Formality": false, "NAME": "lambdatests"}]}
```

Send message

We invoke the *Send Message* function and send a new message to the newly created room. We get the message ID in return, and as we can see, a new message has been added in the DynamoDB MESSAGES table, with a matching ID.

POST ▼ <https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/sendMessage>

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL Text ▼

```
1 {
2   ... "Sender": "John",
3   ... "ROOM_ID": "lambdatests",
4   ... "Text": "lambda tests message"
5 }
```

Body Cookies Headers (5) Test Results Status: 200 OK Time: 1180 ms Size: 192 B

Pretty Raw Preview Visualize Text ▼ 🔍 🗑

```
1 GaXEkiLIAMEVPA=
```


Attribute name	Value	Type
ID - Partition key	GaXEkiaLIAMEVPA=	String
ROOM_ID	lambdatests	String
Sender	John	String
Text	lambda tests message	String
Time	1686583427309	Number
Translations	Insert a field ▼	Map
en	Null	Null
es	pruebas lambda mensaje	String
fr	Null	Null
pl	Null	Null

Get latest messages

Finally, we invoke the *Fetch Latest Messages* function in the newly created room, and as we can see, our message appears in return.

GET
https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/joinFetch?RoomID=lambdatests
Send

Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
✓	RoomID	lambdatests			
	Key	Value	Description		

Body
Cookies
Headers (5)
Test Results
Status: 200 OK Time: 912 ms Size: 403 B Save as Example

Pretty
Raw
Preview
Visualize
Text

```

1 [{"items": [{"Translations": {"en": null, "es": "pruebas lambda mensaje", "fr": null, "pl": null}, "ROOM_ID": "lambdatests", "Text": "lambda tests message", "Time": "1686583427309", "ID": "GaXEkiaLIAMEVPA=", "Sender": "John"}]}]
```

Get recent messages

We also invoke the *Get Room Messages* function, and as expected, the result is the same as above with *Fetch Latest Messages*.

AWS / <https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/getMessages?RoomID=abc>

Save

GET <https://ek5ajs509b.execute-api.us-east-1.amazonaws.com/getMessages?RoomID=lambdatests> Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	RoomID	lambdatests			
	Key	Value	Description		

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 164 ms Size: 403 B Save as Example

Pretty Raw Preview Visualize Text

```
1 [{"items": [{"Translations": {"en": null, "es": "pruebas lambda mensaje", "fr": null, "pl": null}, "ROOM_ID": "lambdatests", "Text": "lambda tests message", "Time": "1686583427309", "ID": "GaXEkiALiAMEVPA=", "Sender": "John"}]}]
```

Functional requirements

Account creation

Users of the application can create accounts and log in.

Users can create an account straight from the landing page of the website. After supplying appropriate data, they will be sent a confirmation email to activate their accounts. After activation, users are able to log in, and access the rest of the site.

Create your account

Username:


Email address:

Password:

Repeat password:

[SIGN UP](#)

Already registered? [Sign in](#)

 **no-reply@verificationemail.com**
do mnie ▾

🌐 angielski ▾ > polski ▾ [Przetłumacz wiadomość](#)

Please click the link below to verify your email address: [Verify Email](#)

Room creation and room joining

Users can create, join and change chat rooms.

Users can access the room interface by clicking the “Rooms” link on the top bar of the page. The interface is split between the left hand side room list, and the right hand side room creation interface. Pressing one of the enter buttons in the list redirects to the chat interface covered in the *Sending and receiving messages* section. Creating a new room takes the room name, the formality setting. After typing the appropriate name and pressing the create room button, the user is redirected to the newly created room, further covered in the *Sending and receiving messages* section.

The screenshot shows the Chat Babel interface. At the top, there is a navigation bar with 'Chat Babel', 'Login', 'Rooms', and 'Logout'. On the right, the user 'kacper' is logged in, and the language is set to 'English (English)'. The main content area is divided into two panels. The left panel, titled 'Join an existing room:', contains a list of room names: 'zleparametry', 'abc', 'asdasdasda', 'Example', 'Example2', 'NewRoom', 'testyopwji', 'AAAAAAAAAA', 'PerfTest', and 'Example3'. Each name has a blue 'Enter' button to its right. A 'Refresh' button is at the bottom of this list. The right panel, titled 'Or create a new one:', has a 'Room name:' input field, a checkbox for 'Formal', and a blue 'CREATE ROOM' button.

Sending and receiving messages

Users can send messages and receive messages from other users in real time.

When joining a room, the user is displayed the chat interface. By typing the message into the text box and pressing send, a message is sent to the server, and is then also eventually displayed for all users in the room in which the message was sent. Additionally, the message is highlighted for the user that sent it.

The screenshot shows the chat interface for the 'Room: asdasdasda'. The room name is displayed at the top in blue. The chat area contains two messages. The first message is from 'Gimigimi:' and says 'Hola'. It has a green 'TRANSLATE' button below it. The second message is from 'kacper:' and says 'Hello!'. It also has a green 'TRANSLATE' button below it. At the bottom of the chat area, there is a text input field containing 'Hello!' and a blue 'Send message' button.

Display of recent messages

Upon joining a room, the application displays the most recent messages that were sent before the join.

25 most recent messages sent before joining are displayed in the chat interface upon joining, regardless of their age.

Room: PerfTest

John:
hello world

TRANSLATE

John:
hello world

TRANSLATE

Translation of messages

Users can translate messages sent by other users to their preferred language by pressing a button.

Messages can be translated by pressing the green translate button in the message body. If the message already has a translation for the selected language (for example requested by another user) then it is also displayed under the message in the place of the translate button.

Gimigimi:
Otro mensaje más

TRANSLATE

Gimigimi:
Otro mensaje más

One more message

Non-functional requirements

Performance

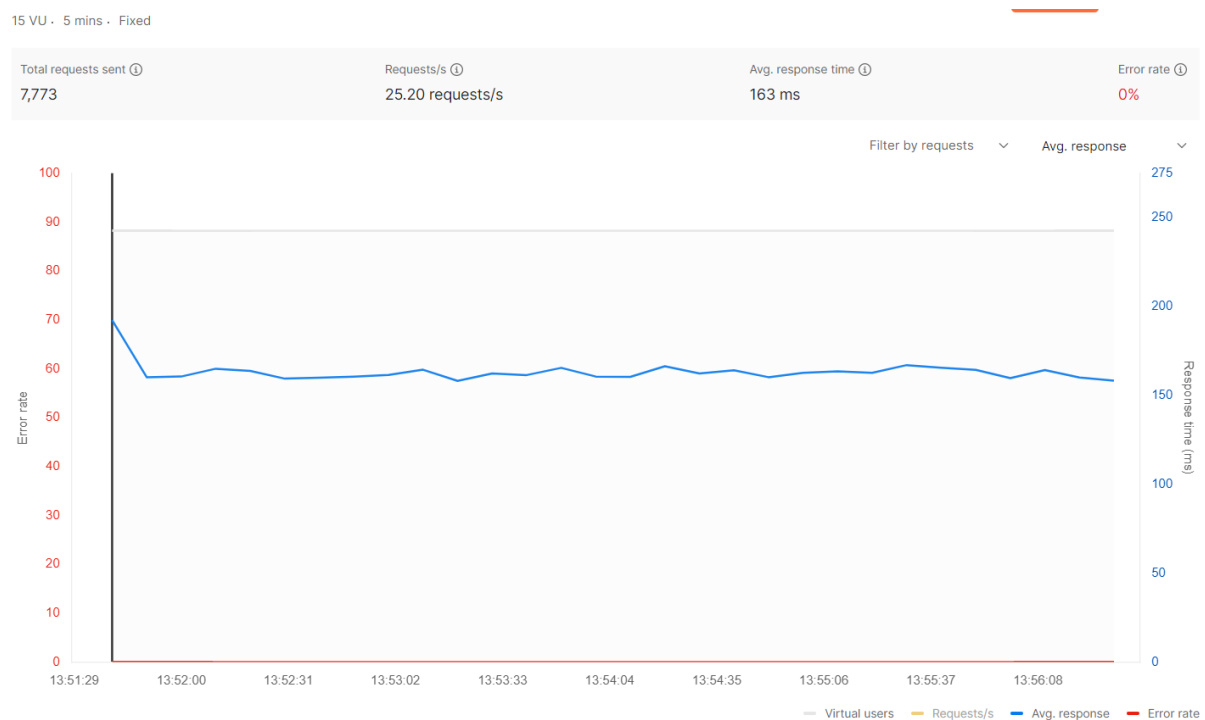
Joining a room and loading previously sent messages in a room does not take more than 1 second, 99% of the time, 95% of messages are delivered within 1.5 seconds.

Send Messages and *Get Messages* services were tested utilizing the Postman's **performance test** utility. Two tests were set up in total:

- Test simulating 15 concurrent users using the *Get Messages* service from 3 separate rooms (test lasted for 5 minutes)
- Test simulating posting new message and getting list of latest messages – repeated 100 times. Results were saved as a *json* file and processed with *python*.

In practice, those tests involved invoking the application API multiple times. Results for first test response times:

Request type	Mean Time	Max Time	Count	# < 1 sec	% < 1 sec
Get messages	160.33 ms	226 ms	265	265	100%



Results for second test:

Request type	Mean Time	Max Time	Count	# < 1 sec	% < 1 sec
Send Messages	229.69ms	511ms	100	100	100%

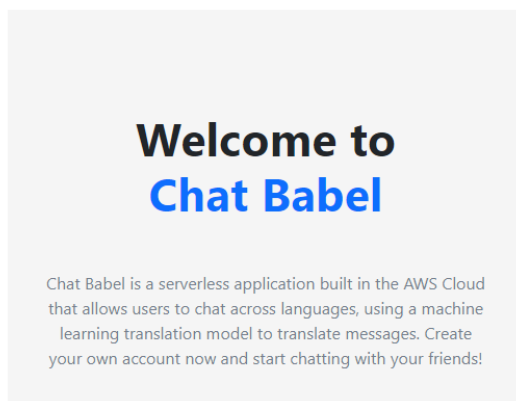
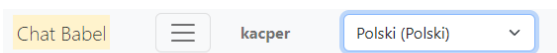
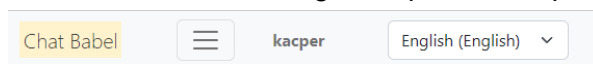
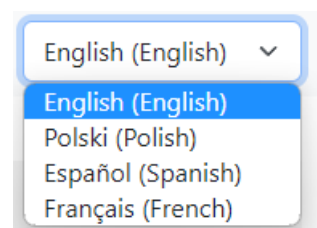
Get Messages	216.07ms	403ms	100	100	100%
Send + Get message	445.76ms	744ms	100	100	100%

It is important to remember, that our app pulls new messages every 0.5s, so that time needs to be added to the results of Send + Get message for the second test - but since all requests take less then 1 second, we still manage to meet the 1.5 second requirement.

Internationalization and proof-of-concept localization

Internationalization of the user interface text and proof-of-concept localization (using machine translation).

The user has access to a dropdown menu which contains 4 different languages. When a different language is selected, the page is automatically updated with the translated text. All human readable texts are stored in a central repository file, which is then machine translated using a script and DeepL API.



Disaster Recovery

Weekly backups of all persistent data set up for disaster recovery.

Automatic backups of all tables in the DynamoDB has been set up (this includes ROOMS and MESSAGES tables). The settings for this backups can be seen on the

Summary

Backup rule name
Messages_backup_rule

Frequency
Weekly
At 05:00 AM UTC, only on Sunday

Start within
8 hours

Complete within
7 days

Lifecycle
Never transition to cold storage
Expire after 5 weeks

right – as we can see, backups take place on every sunday and expire after 5 weeks. So far, 8 backups are available (as 4 weeks passed since the rule has been set up, and each backup contains two tables).

Backups (8) [Info](#)
[Schedule automatic backups](#) and [view backup job details](#) in AWS Backup

[Refresh](#) [View details](#) [Restore](#) [Copy](#) [Delete](#) [Create backup](#)

Find backups by ARN or name

<input type="checkbox"/>	Name	Table	Status	Creation time	ARN	Size
<input type="checkbox"/>	078aa634-1950-4166-...	ROOMS	Available	June 11, 2023, ...	arn:aws:backup:us-east-	252 bytes
<input type="checkbox"/>	7071516c-2a63-41b9-...	MESSAGES	Available	June 11, 2023, ...	arn:aws:backup:us-east-	13,2 kilobytes
<input type="checkbox"/>	87a16a2c-7554-4ab8-...	ROOMS	Available	June 4, 2023, 0...	arn:aws:backup:us-east-	156 bytes
<input type="checkbox"/>	bc7e2b39-78b6-49d5-...	MESSAGES	Available	June 4, 2023, 0...	arn:aws:backup:us-east-	7,3 kilobytes
<input type="checkbox"/>	7b487c73-3c2d-44ff-a...	MESSAGES	Available	May 28, 2023, ...	arn:aws:backup:us-east-	4,3 kilobytes
<input type="checkbox"/>	a4dfcff7-8478-411c-8...	ROOMS	Available	May 28, 2023, ...	arn:aws:backup:us-east-	156 bytes
<input type="checkbox"/>	0db168fa-9fe0-4d28-...	ROOMS	Available	May 21, 2023, ...	arn:aws:backup:us-east-	0 bytes
<input type="checkbox"/>	612caa35-45cc-4b29-...	MESSAGES	Available	May 21, 2023, ...	arn:aws:backup:us-east-	164 bytes

Efficiency

Repeat translations (i.e. requested for the same language by two different users) do not call the translation API multiple times.

This requirement was achieved by storing the translations along with the original message in the DynamoDB database. Subsequent calls to the translation API will return the database record translation, rather than calling the translation API again. Below is the code responsible for that requirement and an example database record, carrying the translation texts.

```

48     text = message["Text"]
49     translated_text = message["Translations"][lang]
50     room_name = message["ROOM_ID"]
51     # If already translated return translated text
52     if translated_text is not None:
53         return {
54             'statusCode': 200,
55             'body': translated_text
56         }

```

ID - Partition key	GJ-nbj4_oAMEVBw=
ROOM_ID	abc
Sender	Gimigimi
Text	Esto es una prueba
Time	1686153980087
Translations	Insert a field ▼
en	This is a test
es	Esto es una prueba

Tuning

Translation functionality can be tuned to tweak formality levels on a room by room basis.

This requirement was achieved by storing the parameter on a room level in the DynamoDB Rooms table, and passing that parameter to the DeepL API depending on the room the message was sent to. Below is the code responsible and to the right are example database records.

```

formality = get_formality_from_room(room_name)
if formality is None:
    return {
        'statusCode': 400,
        'body': "Message was sent in room that does not exist"
    }
response = requests.post(URL, headers={
    'Authorization': f'DeepL-Auth-Key {KEY}'
},
    data={
        'target_lang': lang,
        'text': text,
        'formality': formality
    })

```

NAME ▲	Formality ▼
AAAAAAAAAA	false
abc	false