HiLingual Server

Team 2

Garrett Davidson, Noah Maxey, Nate Ohlson, Riley Shaw

Problem Statement

Our team previously built an iOS app which relied on a backend written by a previous team member. This app is meant to facilitate learning new languages through talking to native speakers. This server has many limitations, including platform compatibility, speed, and scalability. We would like to scrap that server and start over using different technologies to address these issues. We will not be reusing any of the code because we are transitioning from a Dropwizard based Java server to a fully Swift server. We do not intend to use the app client for any part of this course, the client will help guide the high level design of the new server. Because we do not plan to use the client for any part of this project, we will only be submitting tests for the server. Creating this new server will also allow us to design from the ground up with testing in mind to prevent regressions as we add more features in the future.

Background

With our previous server implementation, we had the server hosted remotely, running Java, and relying heavily on the Dropwizard framework. The remote hosting, combined with the resource intensive nature of Java, made keeping the server up an expensive task. Using Java and Dropwizard also limited our flexibility when using our Swift-based iOS client. We would like to convert this server to use Swift, and add some more features to it, to run more efficiently, on our own machines, and be more compatible and extensible with our iOS client.

Environment

This server will be written entirely in Swift 3.0. It will be compiled and run on a Linux host. For storage, we will be a MySQL database for long-term and native Swift storage for short-term storage.

Requirements

Functional:

Backlog ID	**Functional Requirements**	**Hours**
1.✓	As a user, I would like to send and receive photos and audio messages on the server.	**15**
2.	As a user, I would like to translates messages to my native language.	**16**
3.✓	As a user, I would like to store my sent and received Unicode compliant text messages on the server with a maximum 500 characters.	**20**
4.	As a user, I would like to send and receive my flashcards from the server, and edit them.	**20**
5	As a user, I would like to retrieve messages received from a specific user.	**5**
6✓	As a user, I would like to receive and send edits of my profile to the server.	**10**
7	As a user, I would like to be matched with users similar to me. This similarity will be based on the user's known languages and age.	**30**
8✓	As a user, I would like to create a new profile.	**14**
9	As a user, I would like to receive push notifications.	**20**
10✓	As a user, I would like to authenticate using Facebook or Google account.	**10**
	Total:	**160**

Non-Functional:

1. We must have the server run on Linux machines.
2. Server must store all of the data using a MySQL database.

3. Messaging must be as fast and reliable as possible.
4. Messages should be stored securely on the server.
5. Personal/account information must be kept secure.
6. Server must be able to support all target languages.
7. Server must be able to authenticate users via OAuth 2.0.
8. Server must be able to integrate with various support APIs (translation, etc).
9. Server must support a large number of concurrent users.
10. Server must be able to be run locally for testing/development.
11. Server must be very stable and unlikely to crash.

Use Cases

Case: Send a message (text, audio, image)

Action	**System Response**
1. Send POST request with text, audio or picture message	2. Validate message (≤500 character Unicode text or a single valid m4a or PNG file that is less than 10 mb.
	3. Look up the sender by the provided auth token and verify the validity of the auth token
	4. Validate the recipient user
	5. Store message in database
	6. Send successful response code

Case: Request a translation of a received text message

Action	**System Response**
1. Send a translation request by sending a POST request with the id of the message to be translated	2. Verify that the message exists and is a Unicode string ≤500 characters.
	3. Request the translation of the message from the translation server.

	4. Store translated text in database alongside original message.
	5. Send translated text back to user
6. Receive translated message.	

Case: Retrieve messages (text, audio, image)

Action	**System Response**
1. Send GET request with target user ID and optionally the count of messages to return. Default message count is 20 most recent.	2. Look up user id.
	3. Read messages sent to that user id from database.
	4. Respond with messages.

Case: Upload a flashcard set

Action	**System Response**
1. Send POST request with a JSON formatted array of flash cards. Each flashcard object in the array contains two string fields: front and back. Each of these fields can contain a maximum of 50 Unicode characters. The array cannot contain more than 100 elements.	2. Validate and parse the flashcard array.
	3. Store the flashcard set in database
	4. Return successful response code

Case: Retrieve a flashcard set

Action	**System Response**
1. Send GET request with the ID of a flashcard set.	2. Validate auth token and use it to look up user id.

	3. Verify that the requested flashcard set exists.
	3. Retrieve flashcards from database.
	4. Send flashcards to user.

Case: Update profile information

Action	**System Response**
1. Send GET request with profile ID to get current profile information	2. Look up user id
	3. Retrieve user profile information from database
	3. Send profile to user
4. Send POST request with profile info and send differences to server	5. Look up user id
	6. Store changes to profile in database

Case: Request list of matches

Action	**System Response**
1. Send GET request to the match request endpoint	2. Validate auth token and retrieve profile info for user id from database
	3. Run matching algorithm against user's profile information. Matching algorithm is based on the user's age and personal settings for the languages they know and are learning. For user1 to consider user2 a match, user1 must be learning a language that user2 knows, and user1 must know a language that user2 is learning. Matches are sorted by how close they are to your birthday.
	4. Compile list of matching users

	5. Respond with no more than 20 matching users.

Case: Create new profile

Action	**System Response**
1. Send POST request to server with user's valid Facebook or Google auth token.	
	2. Validate the provided auth token with Facebook or Google respectively.
	3. Verify that no user is currently linked to that Facebook or Google account.
	4. Retrieve the user's name and birthdate from Facebook or Google.
	5. Insert the user's name and birthdate into the users table in the MySQL database.
	6. Return the user's id, which is the value of the unique autoincrement "user_id" field of the user's table.
7. The user specifies which language(s) they know and which they would like to learn.	
8. Send a POST request to the server containing a string of comma-delimited ISO 639-1 language codes for both the user's selected known and learning languages.	
	9. Validate the formatting of the languages and that each language listed exists.
	10. Store the strings in the user's profile

	in the database.

Case: Generate push notification

Action	**System Response**
1. Send POST request with message and ID and target ID to server.	2. Validate message and target user ID.
	3. Store message in database.
	4. Generate push notification for recipient.

Case: Authenticate using Facebook/Google

Action	**System Response**
1. POST to server with authentication id and authority token	2. Verify token with authority, by sending the token to Facebook/Google's authorization API.
	3. Create application session token
	4. Send application session token to user