Final Project Report: Bridges

Joel Austin, Jose Zindia, Kinori Rosnow

Introduction:

Bridges Chat Translator is a project for Pitt MSIS class Network and Web Data Technologies. It is a cross-language live-messaging app that enables users that speak two different languages to have their messages auto translated via the azure translation API. This report describes our backend, frontend and database systems. We conclude the report with our roles in this project.

Frontend:

Frontend: EJS/HTML CSS Bootstrap JavaScript

The front end was implemented as follows: We created the login/signup pages using EJS and CSS to enable the users to access the web application. Login and Sign up pages were implemented in EJS and bootstrap. Whenever, the user's login or signup, they were directed to the main dashboard which had the following which had a list with the names of the members on the chat and chat box for sending and viewing messages.

Backend:

Our backend is hosted by a server that handles the user and chat data. The server routes requests and provides requested data back to the client. Sessions are managed by the server. The chat rooms are managed using web sockets that were leveraged from Socket.io.

Here is an example of a route on the server side. This code is used to get messages for a chat room.

```
router.post('/', (req, res) => {
    const currentUser = req.user;
    const roomUser = req.body;
    console.log(req.body.email);
    const roomID = roomUser.email == 'global' ? 'global' : getRoomID([currentUser, roomUser]);

    Message.find(
        {
            "timestamp": {
            $lt: new Date(),
            $gte: new Date(new Date().setDate(new Date().getDate() - 1))
            },
            "room": roomID
        }
    )
    .populate('author', '-password -created_date -_id -email_verified')
    .select('-_id -__v')
    .then((messages) => {
        res.json(messages);
    });
});
```

Sessions are implemented automatically using express-session and a MongoStore.

```
const session = require('express-session');
const MongoStore = require('connect-mongo')(session);
const sessionStorage = new MongoStore({ mongooseConnection: mongoose.connection});
```

Below is an example of a socket.io event listener for when the "message" event is emitted from the client.

```
socket.on('message', async (data) => {
    data.sender = socket.request.user.name;
    data.timestamp = new Date();
    data.original_language = socket.request.user.locale;
    //do translations here for the new message.
    // data.translations = {en-US: 'hello', de-DE: 'gutentaag', ja-JP: "こんにちは"}
    data.translations = await translateAll(data.original_language, data.message);
    data.translations.push({text: data.message, to: data.original_language});
    // Codes are according to RFC 3066 https://tools.ietf.org/html/rfc3066
    // full list: http://www.lingoes.net/en/translator/langcode.htm

    if (!data.room || data.room.email == 'global') {
        data.room = 'global';
    }
    else {
        console.log(socket.request.user.email)
        data.room = getRoomID([socket.request.user, data.room])
    }
    console.log(socket.rooms);
    const newMsg = new Message({
        author: socket.request.user._id,
        timestamp: data.timestamp,
        room: data.room,
        original_language: data.original_language,
        translations: data.translations
    })
    .save()
    .then( async (message) => {
        console.log('----------------------------\nnew message sent:', message);
        await message.populate('author').execPopulate();
        message = message.toObject();
        // remove a bunch of things about the author we don't want to be sent to the client.
        delete message.author.email_verified;
        delete message.author._id;
        delete message.author.password;
        delete message.author.created_date;
        socket.to(message.room).emit('message', message);
    });
```

Database

The database is a MongoDB instance with 3 collections. 1. user 2. sessions 3. messages

The user collection contains the information about the user including their language selection. Passwords are also hashed using bcrypt.

The sessions collection contains information to make the server statefully aware of each user that chats using the app. This is automatically populated with `express-session` and `MongoStore`.

the messages collection contains a record of all the messages sent using the app, as well as the translations into our supported languages.

Accessibility:

Perceivable

We aimed to make font easily readable for users without having to zoom in. For the visually impaired users, all text on the web app is simple so parsers should have no issues. There is not a gaurantee that the chat messages themselves are easy to parse as we cannot control the language of the messages.

Operable

Users can interact with the site in multiple ways such as mouse, keyboard, and touch screen. The buttons are made larger allowing users that may lack finer control to still be able to click the buttons. There is no flashing content so user with photosensitivity to flashing such as certain forms of epilepsy, can use the chat service.

Understandable

Text contrast and font is chosen to be easily visible even for those with colorblindness. Color is not used as an indicator, feedback or information provider. Language is simple and easy to understand. All interactions yield the expected results.

Robust

We are HTML validated. There are no non-standard interface components.

Here is the validation report for the html for our dashboard (the page where you chat with others).

Document checking completed. No errors or warnings to show.

Source

```
<!DOCTYPE html>↩
<html lang="en">↩
  <head>↩
    <meta charset="utf-8">↩
    <meta http-equiv="X-UA-Compatible" content="IE=edge">↩
    <meta name="viewport" content="width=device-width, initial-scale=1">↩
↩
    <title>Bridges</title>↩
    ↩
    <!-- import the webpage's stylesheet -->↩
    <link rel="stylesheet" href="/style.css">↩
    <link rel="stylesheet" type="text/css" href="/css/dashboard2.css">↩
```

Limitations

There are some limitations of the accisibility of the website. Notable we did not have time to create instructional pages that explain the usage of the website. We aimed to make the UI simple enough that it was self explanatory, but some users may find it convenient to have instructions. Another potential improvement is our chats show text as the server recieve and emit it to the chat websockets. As a result the arrival and displaying of messages is at the mercy of the rate at which they arrive. With a sufficiently large chat room or fast enough users, chats may arrive faster than users can read them. Users can scroll but there may be users that would prefer a "slow mode" that allows them to read the chats at a rate that is more accessible to them. Our current answer is the scroll, but some users may prefer a "slow mode".

Documentation of how authentication was implemented passport

Authentication

We implemnted authentication using Passport, which was extended with session storage using MongoStore, which is a node module that stores the session data in our mongoDB. In Bridges, there is actually only one function for both signup and login. This is because both are required to verify that a user with the specified email exists, or does not exist. If it does exist and they are trying to signup, the user is sent to an error screen. If the user doesn't exist and they are trying to login, then they are also sent to an error screen. Below is the signup/login service using passport.

```
passport.use(
    new LocalStrategy({ usernameField: 'email', passReqToCallback: true}, (req, email, password, done) => {
        User.findOne({email: email})
        .then(user => {
            if (!user) {
                let name = req.body.name;
                let locale = req.body.locale;
                if (name == null || locale == null) {
                    throw 'One or more values are missing';
                }
                // create new user, hash the password, and save it.
                const newUser = new User({ email, password, name, locale });
                console.log('creating new user');
                bcrypt.genSalt(10, (err, salt) => {
                    bcrypt.hash(newUser.password, salt, (err, hash) => {
                        if (err) {
                            throw err;
                        }
                        newUser.password = hash;
                        newUser
                        .save()
                        .then(user => {
                            console.log(user);
                            return done(null, user);
                        })
                        .catch(err => {
                            return done(null, false, {message: err});
                        })
                    })
                })
            }
            else {
                // user exists, compare hashes of passwords, then log user in if correct.
                bcrypt.compare(password, user.password, (err, passwordsMatch) => {
                    if (err) {
                        throw err;
                    }
                    if (passwordsMatch) {
                        return done(null, user);
                    }
                    else {
                        return done(null, false, { message: 'Email or Password Incorrect.'});
                    }
                })
            }
        })
    })
);
```

Error Handling

In general, our application contains most of it's error handling on the server side surrounding the authentication, however there is also client side form-validation for the login and signup pages.

There are appropriate error catches for when the user is not signed in, for example. Below is a snippet of that code.

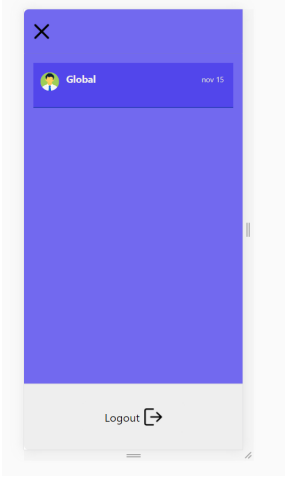
```
const requiresAuth = (req, res, next) => {
    if (req.isAuthenticated()) {
        return next()
    }
    else {
        res.redirect('/auth/login');
    }
}

// if the user is authenticated already, then we forward them to the chat/dashboard.
const forwardOnAuth = (req, res, next) => {
    if (req.isAuthenticated()) {
        res.redirect('/chat');
    }
    else {
        next();
    }
}
```

This snippet contains 2 middleware functions that handle when the user visits a route. If they are not logged in, then they are redirected to the login page. The second function is for if they visit the login page when logged in, then they are forwarded to the chat app.

Mobile/Responsive Design

Bridges Chat Translator is fully functional on mobile screen formats. Below you can see the difference in mobile vs desktop designs.



<

p style='clear: both;'>

As you can see, the mobile format collapses the left side into a hamburger menu that can be toggled.

External Code and Frameworks:

For this project, we ended up using a somewhat minimal spread of external libraries and packages. Most of the libraries we used were for more complex thing such as user authentication and session storage. These would not have been possible to reliably implement in the time frame of the project.

The following is a list of all the external libraries we used. - "bcryptjs": "^2.4.3", - "body-parser": "^1.19.0", - "connect-mongo": "^3.2.0", - "cookie-parser": "^1.4.5", - "dotenv": "^8.2.0", - "ejs": "^3.1.5", - "express": "^4.17.1", - "express-session": "^1.17.1", - "mongoose": "^5.10.3", - "node-fetch": "^2.6.1", - "passport": "^0.4.1", - "passport-local": "^1.0.0", - "passport.socketio": "^3.7.0", - "socket.io": "^3.0.0"

Member Roles:

We defined our roles based on ownership of functionality. We started the project with specific roles, but as the project progressed and we periodically worked together on calls, the lines of our work were blurred. Below we describe our part of the project.

Kinori Rosnow:

My role for this project was handling the language translations and the language data interactions. A core of the site is the translation of language where we leveraged Microsoft's Azure Translation API to translate to every language we support. The translation capabilities are limited by the Azure Translation API capability. The messages are stored in every language supported to maximize speed efficiency with a tradeoff with space. For the message database schema I had the message all stored in the same dictionary so the backend interactions could all be the same. The original message language was stored in the schema separately. To be specific my front end work was the language selection drop down in the top right of the screen. Most of my work; however was on the back end. I routed the requests for the change of language selection to the server which update the user information `Locale` in the database. The client side would then request the messages in the

new language. My part of the system does have a possible flaw in that if a user's `Locale` in the database differed from their keyboard language, there may be a way to trick the system into storing the wrong original language of the message in the database. Given more time I would be interested in addressing this to combat potential malicious users. Although I'm not sure the full extent of the possible harm yet as the translation API can detect correct languages and our system doesn't depend on the language except for what it displays to the users. We do not think this is a security issue.

Joel Austin

My Role primary for the project was to handle implementing socket.io to handle sending and recieving messages. I also achieved a stretch goal we had that was to implemnt individual rooms where people would chat 1:1. My part included both the server-side and client-side implementations of socket.io.

There's actually an interesting system for how I engineered that work. In order to get rooms to work. I needed to generate unique ID that could be deciphered from some data of each user, so I actually ended up alphabetically sorting the 2 user emails and doing a simple hashing on the concatination of the email strings.

```
function getRoomID(users) {
    users.sort((a,b) => {
        return a.email.localeCompare(b.email);
    });
    const [first, second] = users;
    const hash = generateHash(first.email + second.email) + 'A';
    return hash;
}

function generateHash(string) {
    var hash = 0;
    if (string.length == 0)
        return hash;
    for (let i = 0; i < string.length; i++) {
        var charCode = string.charCodeAt(i);
        hash = ((hash << 7) - hash) + charCode;
        hash = hash & hash;
    }
    return hash;
}
```

I also worked on implementing the authentication on the backend for the site using passport.

Jose Zindia

For this project, we worked on almost every bit of the project, but I mainly focused on the front-end user interface: My work mainly focused a lot on the design and usability of the web app. I used EJS and CSS/JS to create and render the following HTML pages: - The Login and sign-up page: To allow user sign up and login. - The chat dashboard: To allow users to access - I also worked on the user model and the server to get the app running.