LeTour Guide Handoff

Cameron Bundy, Richard Homan, Joel Wink
December 2, 2024

Purpose

The purpose of this document is to bring the next group working on this project up-to-speed with our groups work this semester (Fall 2024). If you are a Software Engineering 2 team, we will also give you some tips on how to manage your time and what to look forward to (because we weren't given much heads-up). Ask to receive the Concept Description, Software Requirements Specification (SRS), Software Design Document (SDD), Software Project Plan (SPP) and review these items were necessary. Please know that our product is far off from a final product. It throws warnings and errors, and we are well aware of this. Our goal this semester was to create some sort of functional prototype to show off to the LETU Admissions Department as a proof-of-concept that they could invest in. Unfortunately, due to project unknowns, this was never accomplished. This document should shed light on where we left off and why.

The Concept

Review the Concept Description document. Our goal was to create a prototype; research and development. Accept our source code as such. You may decide to go a different direction with our code, which is fine, but use it to your advantage to better understand the problem and solution.

The Project

Review the section 1 and 2 of the SRS. Here we have made a few assumptions about the size of our group and how long the solution will be running. This may change for your scope.

Review section 3 of the SRS. Our UI is 2 by 2 parts. There are two user types of the solution: tour guides and audience members. There should only be a single tour guide per tour by which many audience members are part of. The two pages for each user serve similar functions. Both landing pages provide a place for the user to know that they have arrived at the right place, and the following action that they should take. An audience member can select their tour and press the join button, which brings them to the in-tour page. This page is what plays the audio. A similar model is copied for the tour guide, except that the tour guide can create a group with a name and pressing the create button brings them to the in-tour page. A warning dialog is presented to both users if they press the Leave/End/Back button. You can ignore the optional requirement 3.2.6 as this was not implemented.

Review section 4 of the SRS. Lots of these items will probably change for you. Many of these items were created arbitrarily because our project was mostly of R&D nature to try to find those limits.

Review section 1 & 2 of the SDD. Again, this will change for your project.

Review section 3 of the SDD. The figure in 3.1 shows our architectural model which is (mostly) still valid where we left it off. We have added another container for debugging purposes which is explained in the next section of this document.

The Repository

The entire project is contained within a docker container. To run the container, read the *README.md* file in the repository. Inside the docker container, there are 3 containers. To modify the docker configuration, use the *docker-compose.yml* file in conjunction with the Dockerfile files in each of the container directories. The ReactJS server serves the front end and is found inside the "letour-app" directory. The Janus container, which serves the backend, is found in the "janus-ab" directory. The third container isn't required for the project but helps for debugging. This includes the Janus demo server provided in the Janus repo found in the "janus-ab-demo" directory.

React Container

The Dockerfile for this container simply downloads the required node modules and runs the react app.

Janus Container

If you open the Dockerfile for this container, you will find the script which actively gets Janus from its repo, gets all the dependencies, and builds only the audiobridge plugin and websocket transport. We use websockets to communicate with the front end. The front end uses the npm module "janus-gateway-js" to interface with Janus which uses websockets (as opposed to the REST HTTP API) as its core transport.

Janus Demo Container

This uses the built-in python web server to serve it. Access the audiobridge demo by accessing the demo page, going to Demos >Audiobridge in the top nav bar. The demo automatically connects to the demo room. To configure the auto-created rooms, modify the janus-ab/config/janus.plugin.audiobridge.jcfg file.

Important Notes

Sometimes you will need to shell into a container to do manual work such as reinstall node modules on the front end or do some configuration for the back end. You can use `docker exec -it <container name> bash` to shell into a specified container. Also, if a docker container command returns (such as when react has an irrecoverable error), the container will shut down. There is a line in the docker file `CMD ["sleep", "infinity"]` that is commented out and you can uncomment to keep the container open while you fix it.

Where We Left Off

At the end of our group efforts, we were able to complete the front- and back-end functionality. However, a big issue for us was trying to get the clients on the same network to communicate with each other. We found that, because of the complexity of the LETU 'letnet' network configuration and how different browsers act, it was hard to reproduce problems or identify a common symptom.

Connectivity Issues

At the very core of WebRTC, which is the functionality that powers Janus, the PeerConnection instance controls the connection between the client and other clients (or in the

audiobridge case the server). The web browsers themselves define the behavior the PeerConnection which can differ between implementations. Some web browsers will provide an mDNS address as the local description which is not identifiable on the 'letnet' network. A possible remedy for this is to host the solution on a custom-made local network hotspot without mDNS. This could be implemented on a Raspberry PI. It would have to host the network, host a DNS service, and could also host the server.

Security Issues

Because all connections with any service on the server is not secure, many web browsers won't let the user use WebRTC functionality or even access the site at all without changing some browser flags. This is, of course, not something we want our users to do and is usually impossible in a mobile environment anyway. The front-end server would need to run HTTPS and have a certificate signed by a certificate authority that is recognized by most mobile operating systems.