Portland State University: CS594
June 4, 2021
Intended status: IRC Class Project Specification
Expires: December 2022
Brita Budnick and Michael Howard

# Internet Relay Chat Class Project

**Project repository:** https://github.com/zemar/cs594-project

**Launched application:** http://35.247.113.187/

## Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. This document may not be modified, and derivative works of it may not be created, except to publish it as an RFC and to translate it into languages other than English.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet- Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on December 1, 2022

## Copyright Notice

**Abstract**

This memo describes the communication protocol for an IRC-style client/server system for the Internetworking Protocols class at Portland State University.

**Table of Contents**

## 1. Introduction

This document describes the implementation of an Internet Relay Chat protocol. We provide a collection of predetermined client-server protocols to route all necessary messages needed for internet relay chat among clients. A central server relays messages to all clients. A database stores the messages transmitted from each chatroom, a master list of current chatrooms, members of each chatroom, and base 64 encoded files transmitted by users. Users can join a room, join multiple rooms, message a single room, message multiple rooms, send a file in a room and leave a room. Any message a user sends becomes visible to all other members of that chat room.

Developing a chat client/server application can be simplified if the communication between them is cleanly structured and takes advantage of hardened application and transmission protocols. Thus ChatMP was developed to provide an easy messaging framework which client/server developers can easily integrate.

## 2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.

## 3. Basic Information

The ChatMP protocol depends on HTTPS and TCP/IP. HTTPS must be at least v1.1 and will be handled in the client application (webpage). The client was developed using React and the server was developed using Node.js. The messaging protocol is inherently asynchronous; it takes advantage of React lifecycles to asynchronously request information from the database and serve information to the user.

All messages go over HTTP(s) and thus take advantage of predetermined handshaking for GET/POST/PUT/DELETE methods. TLS termination of the connection is supported to provide end-to-end encryption and each connection is transmitted using TCP thus enabling traffic congestion handling and retransmission.

Iterations of the development of each protocol rely on GitHub Actions which was used to update workflows implemented in the repository.

Each version of the codebase was published to the Google Container Registry and deployed to Google Cloud in a Google Kubernetes Engine (GKE) environment. This occurs through an automated CI/CD pipeline hosted through GitHub Actions. A shared cluster in MongoDB was established to host a non-relational database to be queried by the server.

## 4. Message Infrastructure

### 4.1. Generic Message

In src/backend/api/update.js:

```
const  { dbUpdate, dbChat } = require('../db');

const update = async (request) => {
    let existing = await dbChat(request)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbChat failed', err) });

    let updatedChat = [];
    if (existing[0].chat) {
        updatedChat = existing[0].chat.concat([request.chat]);
    } else {
        updatedChat[0] = request.chat;
    }

    let updatedRequest = {'id': request.id, 'chat': updatedChat };
    console.log(`updated request ${JSON.stringify(updatedRequest)}`);
    let result = await dbUpdate(updatedRequest)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbUpdate failed', err) });

    return result;
};
```

In src/frontend/components/MessageBox.js:

```
let MessageBox = (props) => {
    let multisend = false;

    const sendMessage = (e) => {
        e.preventDefault();
        let request = {
            'id': props.currentRoomId,
            'chat': {
                'user': props.currentUser,
                'message': e.target.message.value
            }
        };
        if(multisend) {
            props.multiUpdate(request);
        } else {
            props.updateRoom(request);
        }
    }
```

4.1.1. Field definitions

- The MessageBox function is defined in a MessageBox component on the DOM tree. It accepts a property object as props that contains the message the user just submitted.
- The multisend boolean denotes whether the user has selected multiple chatrooms to send their message to or if the message will only be sent to a single (the current) chat room.
- The sendMessage function is triggered on a click event from the user. When this function is invoked, the request JSON object's fields are updated.
- Finally, if the multisend boolean is true, the request object is passed to the multiUpdate field of the props object. If this boolean is false, it's passed to the updateRoom field of the props object.
- updatedRequest - an object containing id and chat that is passed to the dbUpdate function which updates the chat log in Mongo

4.2 Error Messages alert

In src/frontend/components/App.js:

```
const createRoom = (request) => {
        apiCreate(request).then(response => {
            if (!response.result.ok)
                alert(`Create API not currently available`)
            else if (!response.result.n)
                alert(`Not able to create ${request.title}`);
```

```
            else {
                setLoading(true);
                apiList().then ( chatrooms => {
                    setList(chatrooms);
                    setLoading(false);
                });
            }
        });
    }
```

4.2.1. Usage

If a user attempts to join or leave a chat room, create a new chat room, get an updated version of the chat room messages, delete a room, or send a file, that attempt triggers a request to the server which queries the database. If that process results in an error code, the user MAY be presented with an alert that informs them of the error.

4.2.2. Field definitions:

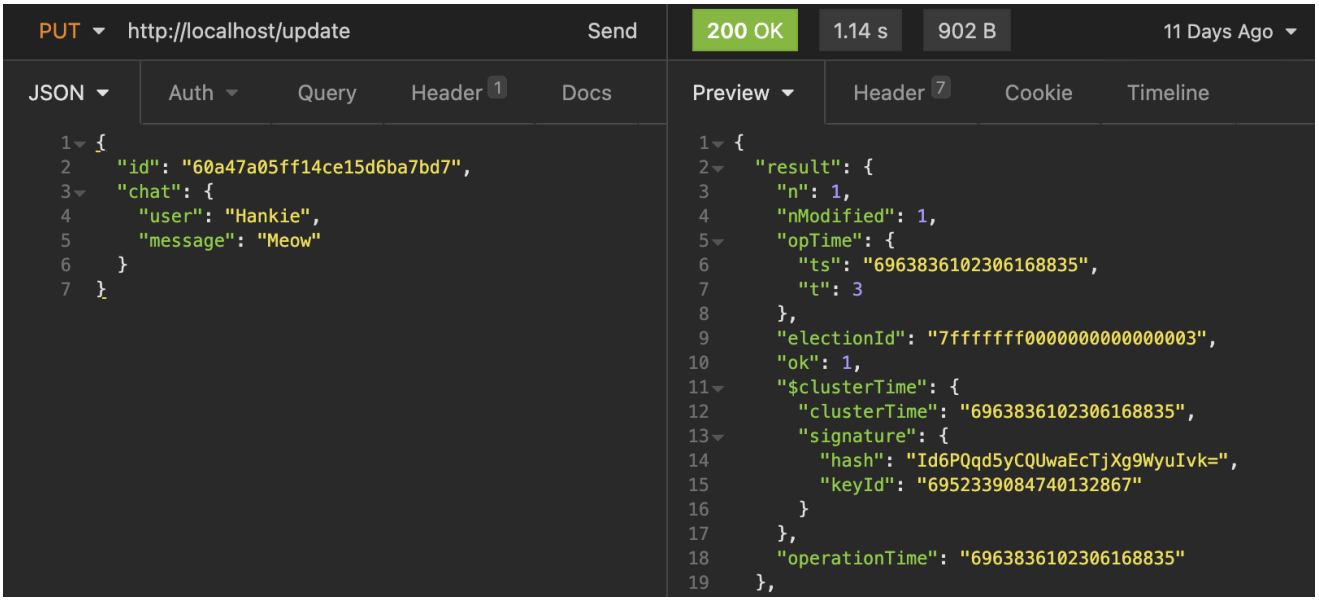
- response.result.ok - notification of whether or not the request was successful
- response.result.n - notification that the user's input led to an unsuccessful API call

4.2.3. Error Codes

- 200: successful API call
- 400: client error
- 500: server error

**5. Label Semantics**

Any data transmitted between the client and server is packaged as a JSON object payload. Each of these objects correspond to the expected formatting in the Mongo database. For example, if the update request included the following information to be committed to the database, the server would relay a PUT request to the database with the JSON object it had received from the client. The appropriate database would then be updated or, if appropriate, return an error message.




The header for each request is similar, but an informative example is provided by the GET request. Insomnia [11], a useful tool that provides analytic information on the details of a request, provides this insight into the GET request:

```
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 276
ETag: W/"114-emmyMTYIDP0Zys2CuWaOILh9Jwg"
Date: Sun, 30 May 2021 22:42:28 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```

### 6. Client Messages

6.1. First message sent to the server (PUT)

In src/backend/router.js:

```
router.put('/update', async (req, res) => {
    try {
        let response = await api.update(req.body);
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

In src/frontend/components/App.js:

```
const App = () => {
```

```
    const [list, setList] = useState([{'id':'0', 'title':'Initial
room'}]);
    const [currentUser, setCurrentUser] = useState('roomadmin');
    const [currentRoomId, setCurrentRoomId] = useState('0');
    const [currentRoom, setCurrentRoom] = useState({'id':'0',
'users':['roomadmin'], 'title':'Inital room',
'chat':[{'user':'roomadmin', 'message':'hello'}]});
    const [loading, setLoading] = useState(false);
    const [pinnedRooms, setPinnedRooms] = useState([]);

    // Runs on 1st render only
    useEffect(() => {
        let roomId = localStorage.getItem("currentRoomId");
        let user = localStorage.getItem("currentUser")
        roomId ? setCurrentRoomId(roomId) : null;
        user ? setCurrentUser(user) : null;
        setLoading(true);
        apiList().then ( chatrooms => {
            setList(chatrooms);
            setLoading(false);
        });
    }, []);
```

6.1.1. Usage

When a user first navigates to the Felix Chat application or when they create a new chat room, the chat room is populated with a default state that includes an id, users list with a roomadmin username, title, and chat greeting from the roomadmin. The useEffect() function initiates the first render of the application.

6.1.2. Field Definitions

- const App = () => {} - defines the topmost function in the DOM hierarchy
- const [list, setList] - a Hook that is an array with two values that are destructured to hold and set the list value
- const [currentUser, setCurrentUser] = useState('roomadmin'); - a Hook that establishes the state of the user variable as roomadmin.
- const [currentRoom, setCurrentRoom] = useState({'id':'0', 'users':['roomadmin'], 'title':'Inital room', 'chat':[{'user':'roomadmin', 'message':'hello'}]}); - a Hook that establishes each value in the currentRoom object to default values.

6.2. Listing Rooms (GET)

In src/backend/router.js:

```
router.get('/list', async (req, res) => {
    try {
        let response = await api.list();
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

In src/backend/api/list.js:

```
const  { dbList } = require('../db');

const list = async () => {

    let result = await dbList()
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbList failed', err) });

    return result.map( item => { return {'id':item._id,
'title':item.title}; });
};

module.exports = list;
```

In src/frontend/components/SideBar.js:

```
let SideBar = (props) => {
    let lineItem = "";
    if(props.chatrooms) {
        lineItem = props.chatrooms.map(({ users, id, title }, index) =>
            <LineItem key={index} id={id} title={title}
currentUser={props.currentUser}
             setRoomId={props.setRoomId} joinRoom={props.joinRoom}
leaveRoom={props.leaveRoom}
             deleteRoom={props.deleteRoom} setPinned={props.setPinned} />
        )
    }
    return (
        <aside className='sidebar'>
            {lineItem}
        </aside>
    )
};
```

6.2.1. Usage

The apiList function, referenced in the code excerpt included in section 6.1, is called in the useEffect function which is executed on the first render of the application. The apiList function is also called whenever the list of chatrooms needs to be updated due to interaction from the user (on create or delete). To adhere to the modularity expected in a React application, each component manages a specific function or group of functions with specific functionality. The list of chatrooms is displayed in the side bar area of the application so the majority of chatroom list functionality is in the SideBar component.

When the chatrooms need to be updated and the list of current chatrooms need to be displayed, the list object is passed down the component hierarchy to the SideBar function component. In that component, the users, id, and title fields in the props.chatrooms object are mapped to properties in a lineItem component. The LineItem function component is responsible for handling changes to the list of chatrooms, including when the user clicks the name of a chatroom, when a chatroom is joined, when a user leaves a chatroom, when a user deletes a chatroom, or when a user pins a chatroom to facilitate joining multiple chatrooms.

#### 6.2.2. Response

The server returns a list of chatrooms as a JSON object.

### 6.3. Joining Rooms (PUT)

In src/backend/router.js:

```
router.put('/join', async (req, res) => {
    try {
        let response = await api.join(req.body);
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

In src/backend/api/join.js:

```
const join = async (request) => {
    let existing = await dbChat(request)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbChat failed', err) });

    let updatedUsers = [];
    if (existing[0].users) {
        if (!existing[0].users.includes(request.user)) {
            updatedUsers = existing[0].users.concat([request.user]);
        } else {
            updatedUsers = existing[0].users;
        }
    } else {
        updatedUsers[0] = request.user;
```

```
    }

    let updatedRequest = {'id': request.id, 'users':updatedUsers};
    let result = await dbJoin(updatedRequest)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbJoin failed', err) });

    return result;
};
```

In /src/frontend/components/App.js:

```
    const joinRoom = (request) => {
        apiJoin(request).then(response => {
            if (!response.result.ok)
                alert(`Join API not currently available`)
            else if (!response.result.nModified)
                alert(`${request.user}, you are already a member`);
            else {
                let request = {'id': currentRoomId};
                setLoading(true);
                apiChat(request).then ( chatroom => {
                    setCurrentRoom(chatroom);
                    setLoading(false);
                });
            }
        });
    }
```

6.3.1. Usage

Each chatroom name rendered in the sidebar area of the application is clickable. When it's clicked, a dropdown menu appears with the option to join that room. That line item click change is passed back up the component hierarchy to App.js and the joinRoom function referenced in the above section 6.3. is called.

6.3.2. Field Definitions

- apiJoin - this function accepts the request variable which holds the name of the chatroom the user has requested to join
- response.result.ok - this response code from the server provides feedback on whether or not the request to join a room was successful
- response.result.nModified - checks to make sure the user isn't trying to join a room they are already a member of
- apiChat(request) - the last option in apiJoin calls an anonymous function setting the chatroom name that was clicked to the current chatroom value

6.4 Deleting a Room (DELETE)

In src/backend/router.js:

```
router.delete('/delete', async (req, res) => {
    try {
        let response = await api.del(req.body);
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

In src/backend/api/delete.js:

```
const  { dbDelete } = require('../db');

const del = async (request) => {

    let result = await dbDelete(request)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbDelete failed', err) });

    return result;
};

module.exports = del;
```

In src/frontend/components/Api.js:

```
export const apiDelete = async (request) => {
    let result = await fetch('/delete', {
        headers: {
            'Content-Type': 'application/json'
        },
        method: 'DELETE',
        body: JSON.stringify(request)
    })
    .then(res => {
        if(res.status >= 300) { throw new Error(res.statusText); }
        return res.json();
    })
    .catch(err => {
        console.log(err);
    });
    return result;
```

```
};
```

In src/frontend/components/App.js:

```
const deleteRoom = (request) => {
        apiDelete(request).then(response => {
            if (!response.ok)
                alert(`Delete API not currently available`)
            else if (!response.deletedCount)
                alert(`Room has not been deleted`);
            else {
                setLoading(true);
                apiList().then ( chatrooms => {
                    setList(chatrooms);
                    setLoading(false);
                });
            }
        });
    }
```

In src/frontend/components/LineItem.js:

```
 const handleClickDelete = (e) => {
        e.preventDefault();
        if (!`${props.id}`) {
            alert(`No valid chat room id`)
        } else {
            let request = {
                'id': props.id
            };
            props.deleteRoom(request);
        }
    };
```

6.4.1. Usage

When the user clicks the name of a chatroom in the side bar area of the application, the option to delete a chat room is listed in a drop down menu. When the user clicks the delete button on a chatroom, that click triggers a LineItem event where the id of the chatroom is checked, and if it's valid, that room id is passed back up the component hierarchy to App.js to be deleted. From App.js, apiDelete is called with the id of the room to be deleted.

6.4.2. Field Definitions

- handleClickDelete - on click event, the id of the room that was clicked is checked and if an id for that chatroom exists in the props, then the id is passed as the parameter of a request object to the deleteRoom function.

- deleteRoom - expects to forward a request to the apiList function if the response from the server is successful
- apiDelete - fetches the delete route with a stringified JSON object that is request

6.5. Leaving a Room (PUT)

In src/backend/router.js:

```
router.put('/leave', async (req, res) => {
    try {
        let response = await api.leave(req.body);
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

In src/backend/api/members.js:

```
const members = async (request) => {

    let result = await dbMembers(request)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbMembers failed', err) });

    return result[0].users;
}

module.exports = members
```

In /src/frontend/components/LineItem.js:

```
    const handleClickLeave = (e) => {
        e.preventDefault();
        if (!`${props.currentUser}`) {
            alert(`Please provide your username before leaving.`)
        } else {
            let request = {
                'id': props.id,
                'user': props.currentUser
            };
            props.leaveRoom(request);
        }
    };
```

In /src/frontend/components/App.js:

```
const leaveRoom = (request) => {
        apiLeave(request).then(response => {
            if (!response.result.ok)
                alert(`Leave API not currently available`)
            else if (!response.result.nModified)
                alert(`${request.user}, not able to leave`);
            else {
                let request = {'id': currentRoomId};
                setLoading(true);
                apiChat(request).then ( chatroom => {
                    setCurrentRoom(chatroom);
                    setLoading(false);
                });
            }
        });
    }
```

6.5.1. Usage

In a pattern similar to joining a room, when the client clicks a chatroom name in the side bar area of the application, an option on the dropdown menu that is clicked includes the option to leave the room. In LineItem.js, the user must input their username before they leave so that they can be removed from the list of current users occupying that room. The action is passed back up the component hierarchy to the topmost component where a leaveRoom function is called with the request which holds the name of the chatroom that had been clicked. Two errors are possible, but if the server does not respond with a code indicating failure, the request receives the value of the currentRoomId and the apiChat function updates the list of users in the room the user just clicked to leave.

6.5.2. Field Definitions

- request - an object with two fields, id and currentuser
- leaveRoom - a function that handles errors from the apiLeave function and, on success, calls the apiChat function to update the chatrooms.

6.6. Sending Messages (PUT)

In src/backend/router.js:

```
router.put('/update', async (req, res) => {
    try {
        let response = await api.update(req.body);
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
```

```
});
```

In src/backend/api/chat.js:

```
const  { dbChat } = require('../db');

const chat = async (request) => {

    let result = await dbChat(request)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbChat failed', err) });

    return result[0];
};

module.exports = chat;
```

In src/frontend/components/MessageBox.js:

```
    const sendMessage = (e) => {
        e.preventDefault();
        let request = {
            'id': props.currentRoomId,
            'chat': {
                'user': props.currentUser,
                'message': e.target.message.value
            }
        };
        if(multisend) {
            props.multiUpdate(request);
        } else {
            props.updateRoom(request);
        }
    }
```

6.6.1. Usage

When the user first joins a chatroom, there is a form with a Submit button where a message can be submitted. Setting a username is a prerequisite to joining a chatroom, so their outgoing messages are automatically paired with their current username.

6.6.2. Field Definitions

- request - an object with id, chat, user, and message fields
- multisend - a variable that captures whether or not a user wants to send messages to multiple rooms
- updateRoom - a function that expects the request as parameter so that the chat feed

can be updated in the application

### 6.7. Creating Rooms (POST)

In src/backend/router.js:

```
router.post('/create', async (req, res) => {
    try {
        let response = await api.create(req.body);
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

In src/backend/api/create.js:

```
const  { dbCreate } = require('../db');

const create = async (request) => {
    if (request.title == undefined) { request.title = "This is my test
chat room"; }

    let result = await dbCreate(request)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbCreate failed', err) });

    return result;
};

module.exports = create;
```

In src/frontend/components/App.jssrc/frontend/components/App.js:

```
    const createRoom = (request) => {
        apiCreate(request).then(response => {
            if (!response.result.ok)
                alert(`Create API not currently available`)
            else if (!response.result.n)
                alert(`Not able to create ${request.title}`);
            else {
                setLoading(true);
                apiList().then ( chatrooms => {
                    setList(chatrooms);
                    setLoading(false);
                });
            }
        });
```

```
    }
```

6.7.1. Usage

Creating a room follows a similar pattern to joining or leaving a room. The createRoom function handles a request parameter by passing it to the apiCreate function. If that function returns an error response, the room cannot be added to the database of chatrooms. If the server returns a success code, then the apiList function is called and the list of chatrooms is updated to include the newly created room.

6.7.2. Field Definitions

- createRoom - a function at the topmost component of the React hierarchy that handles the user's request to create a new room
- response.result.ok - a variable that indicates what code the server returned when an API request was made to add a chatroom name to the database.
- apiList - a function that sends an updated JSON object with the new chatroom name to the database

6.8. Listing Members (POST)

In src/backend/router.js:

```
router.post('/members', async (req, res) => {
    try {
        let response = await api.members(req.body);
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

In src/backend/api/members.js:

```
const  { dbMembers } = require('../db');

const members = async (request) => {

    let result = await dbMembers(request)
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbMembers failed', err) });

    return result[0].users;
}

module.exports = members;
```

In src/frontend/components/Members.js:

```
let Members = (props) => {

    return (
        <div className='members-container'>
            <ul className="members">
                {props.currentRoom.users.map( user => {
                    return <li key={user}>
                        {user}
                    </li>
                })}

            </ul>
        </div>
    )
};
```

6.8.1. Usage

Members for a chatroom are listed consistent with whatever chatrooms have been joined.

6.8.2. Field Definitions

- currentRoom.users - an object with users properties that is listed

6.9. Adding a file to a room (PUT)

In src/backend/router.js:

```
router.put('/file', async (req, res) => {
    try {
        let response = await api.file(req.body);
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

In src/backend/api/file.js:

```
const file = async (request) => {
    if (request.file == undefined) { request.file = btoa("This is my
test file"); }

    let result = await dbFile(request)
    .then((res) => { return res; })
```

```
    .catch((err) => { return console.log('dbFile failed', err) });

    return result;
};
```

6.9.1. Usage

The user is presented with an “Add File” button. On click, they can choose a file from their personal computer’s file browser, that file is base 64 encoded, and that encoded file is uploaded to the database via PUT.

6.9.2. Field Definitions

- res.json(response) - a JSON object that is the file to be committed to the database
- dbFile(request) - a function call with a request parameter; the request contains the id of the chatroom to which the file is to be added and the contents of the file

**7. Server Messages**

7.1. Listing Response

In /src/backend/api/list.js:

```
const  { dbList } = require('../db');

const list = async () => {

    let result = await dbList()
    .then((res) => { return res; })
    .catch((err) => { return console.log('dbList failed', err) });

    return result.map( item => { return {'id':item._id,
'title':item.title}; });
};

module.exports = list;
```

In /src/backend/db/index.js:

```
let dbList = async () => {
    const client = await dbClient();
    const col = await dbCol(client);
    let results = await col.find({}).toArray()
        .then((res) => {return res; })
        .catch((err) => { console.log('DB find all failed', err); });
    client.close();

    return results;
```

```
};
```

In /src/frontend/components/Api.js:

```
export const apiList = async () => {
    let result = await fetch('/list', {
        headers: {
            'Content-Type': 'application/json'
        },
        method: "GET",
    })
    .then(res => {
        if(res.status !== 200) { throw new Error(res.statusText); }
        return res.json();
    })
    .catch(err => {
        console.log(err);
        let empty = [];
        return empty;
    });
    return result;
};
```

7.1.1. Usage

When a component experiences a change due to user input, it is re-rendered with the appropriate useEffect function. This means any change to the list of users or the messages in any chatroom is automatically rendered to any user. When the apiList function is called, it asynchronously awaits a GET request from the backend list function. Depending on the status returned by that request, the current list of chatrooms is returned to the frontend. If the response is anything other than 200, an error message is returned.

7.1.2. Field Definitions

- status - a code indicating the result of the request to the database
- dbList - queries the Mongo database

7.2. Forwarding Messages to Clients

In src/backend/db/index.js:

```
let dbUpdate = async (request) => {
    const client = await dbClient();
    const col = await dbCol(client);
    const {id, chat} = request;
    let oid = new mongodb.ObjectID(id);
    let results = await col.updateOne({ '_id' : oid}, { $set: {'chat':
```

```
chat}})
        .then((res) => {return res; })
        .catch((err) => { console.log('DB updateOne failed', err); });
    client.close();

    return results;
};
```

7.2.1. Usage

Since the application is re-rendered on any change, messages are not forwarded to clients, but an up-to-date version of the application is served to the client when anything about the application changes. An integral part of this process is the dbUpdate function.

7.2.2. Field Definitions

- oid - a new mongodb object set to the current id consistent with the request.
- updateOne - a function that expects a JSON object with an id and $set as parameters which are dereferenced

7.3. Deleting all rooms

In src/backend/router.js:

```
router.delete('/deleteAll', async (req, res) => {
    try {
        let response = await api.deleteAll();
        res.json(response);
    } catch(err) {
        console.log(err);
        res.sendStatus(500);
    }
});
```

7.3.1. Usage

This functionality is not visible to the user from the client side; it is only for the maintenance of the contents of the database.

7.3.2. Field Definitions

- deleteAll() - a function call that asynchronously can be executed to delete all data held in the Mongo database

**8. Error Handling**

This application includes extensive backend testing. Each path is tested for complete coverage and each request is tested via unit testing with Jest and run in an automated fashion via the CI/CD pipeline. If a connection is unsuccessful, that error is relayed up through the layers of the application and displayed to the user as a useful and informative error message alert. If the connection between the server and client is lost, the user MUST

be notified. If the client detects that their connection to the server has been severed, it MUST register that a disconnection has occurred and MAY attempt to reconnect, probably by the user reloading the page.

**9. "Extra" Features Supported**

The user is able to add a file to a room as described in section 6.10. All messages are transmitted using HTTPS. The application is hosted in a GKE environment. Extensive unit testing is included for the backend portion of the application.

**10. Conclusion and Future Work**

The FelixChat application implements a 4-tier architecture with a Mongo database, a Node.js backend, a middle layer that fields requests between the client and server, and a React client. 10 API calls implemented end-to-end support message relay functionality. Clients could implement their own protocols that would successfully utilize the middle layer, Node.js backend and Mongo database.

Future work could include testing for the React frontend and refactoring implementations of React component hierarchy to avoid prop drilling and to be consistent with functional JavaScript language standards.

**11. Security Considerations**

Messages are transmitted via HTTPS, but the messages themselves are not encrypted. File transfer undergoes base 64 encryption, but there is no criteria that would stop a user from uploading a potentially malicious file. Any client implementation aside from the one provided should implement their own user-to-user encryption protocol.

**12. IANA Considerations**

None.

12.1 Normative References


1. https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi/related
2. https://create-react-app.dev/docs/adding-a-stylesheet
3. https://reactjs.org/docs/thinking-in-react.html
4. https://rangle.io/blog/simplifying-controlled-inputs-with-hooks/
5. https://reactjs.org/docs/lists-and-keys.html#keys
6. https://www.robinwieruch.de/react-pass-props-to-component
7. https://reactjs.org/docs/context.html
8. https://www.youtube.com/watch?v=TNhaISOUy6Q&ab_channel=Fireship
9. https://www.oreilly.com/library/view/learning-react-2nd/9781492051718/
10. https://eclass.teicrete.gr/modules/document/file.php/TP326/%CE%98%CE%B5%CF%89%CF%81%CE%AF%CE%B1%20(Lectures)/Computer_Networking_A_Top-Down_Approach.pdf
11. https://insomnia.rest/


**13. Acknowledgements**


This document was prepared with formatting guidance from the document provided by Professor Nirupama Bulusu for the Internetworking Protocols Course Project.