Contents of this File

Connecting to Server

The server (AWS EC2 instance) has the key pairs of both development machines. Connecting to the server requires the usage of the following command in a terminal:

```
ssh ubuntu@ec2-13-58-143-176.us-east-2.compute.amazonaws.com
```

This is the default SSH connection method as specified by AWS.

Server Workflow

Changes to the server are committed to the *origin* remote branch located on GitHub.com. When the commit is to the master branch from either development computers, git hooks simultaneously push to the *production* remote located on the AWS server using:

```
git push production master
```

A tutorial of this setup is explained in the section **Git Tips**. The EC2 instance has the key pairs of both development machines. Once the server receives the commit, node.js restarts the server process running from the server.js file using the following command:

```
sudo pm2 restart server
```

This functionality can also be created by starting server.js using the *nodemon* utility. At this point, if any 502 responses come back from the server, there was an error in the committed code. Developer should at that point connect to the server (described in the section **Connecting to Server**) and use the following command:

```
sudo pm2 list
```

This will display the currently running node apps. There should be an running by the name of *server*. If an error has occurred it will show "errored" in the status column. Use the following command to find the error:

```
sudo pm2 logs server --lines 100
```

The lines argument can be decreased, increased, or omitted.

Updating Node.js Packages

Inside the ~/messengerAPI/package.json file are the server application's dependencies. See this file for formatting. When adding a new dependency (or in the case if a necessary update) call the following command:

```
sudo npm install
```

Development Machines

To start developing on a local machine, the server.js file from the server repository must be started. Ensure the same packages are installed as matches the server by following the instructions in **Updating Node.js Packages**. The instructions in **Server Workflow** can be followed on the development machine but first use the following command to start the MongoDB database:

```
mongod
```

This will attach itself to a terminal and can be stopped using Ctl+C. After this, start the node server application by changing the terminal's focus directory to the location of the server.js file and then call:

```
sudo pm2 start server.js
```

The location of server.js is located in the server git repository. To test API calls, instead of using https://hm478project.me/[API CALL]/, you would use http://localhost:8080/[API CALL]/.

JWT

The secret used in JWT signing is generated when *server.js* is started and stored in key.json. The JWT header contains the username of its owner. The expiration date for the token is set for 24 hours from creation. A token is given to a user after clearing *login2*. Before using a websocket, our server had middleware code to prevent access to certain API's that required a valid JWT to access. Now the JWT is checked in the open websocket stream.

Server Details

- AWS EC2 Instance
  - Hosts our server.
- Express
  - Listen to traffic on ports and handle it.
- MongoDB
  - The database.
- Mongoose
  - Interacting with MongoDB.
- Node.js
  - JavaScript for web deployment.
- Ubuntu 14
  - Operating system of our server.

• PM2
    - Process manager for node.js. Otherwise node processes are in the list of linux processes running. Pm2 better handles starting/restarting/organizing node processes.
• NPM
    - Package manager for node.js.
• Socket.io
    - Websocket framework.
• socketio-jwt
    - JWT validation for socket.io.

API directory location in the ubuntu user directory:

    ~/messengerAPI/

Password Policies

• Minimum length is 8.
• *(To Implement)* At least 1 number.
• *(To Implement)* At least 1 capital letter.

Database Model

User
    name: String
        User-specified. Unique to every user.
    password: String
        User-specified password concatenated with salt after hashing using HMAC SHA-256.
        Checked at time of register for password policies.
    challenge: String
        Generated when user initiates login1 for remote login.
    salt: Number
        16 byte CSPRNG number generated for each user at time of account registration.

Messenger API

Non-socket calls:

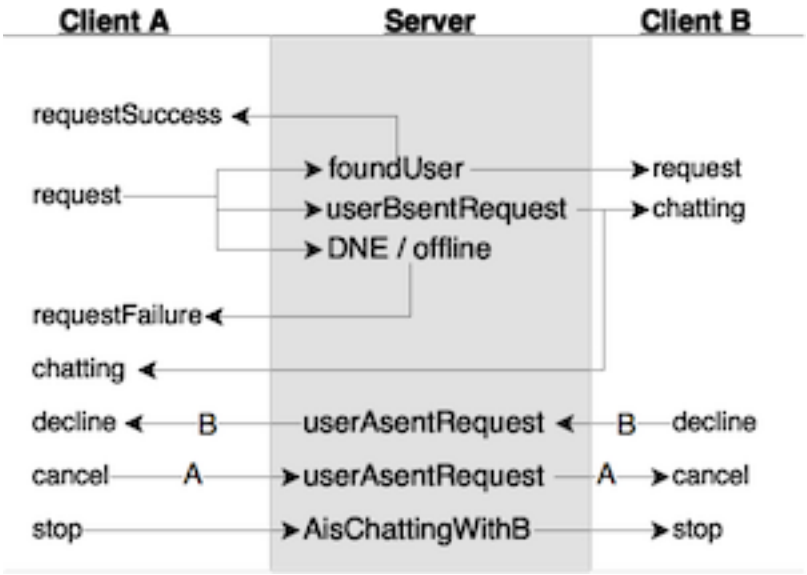
/login1/
    POST
        Keys
            name
    Returns - Salt and Challenge if user exists.

/login2/
    POST
        Keys
            name
            tag
    Returns - JWT if correct tag was produced.
/register/
    POST
        Keys
            name
            password
/users/
    GET
        Returns - All users in database.
            _id: String
            name: String
            password: String
        !!! Used for development. Remove for production.
/users/
    DELETE
        Removes all users.
        !!! Used for development. Remove for production.

Websocket calls:

- To access the websocket, a valid JWT must be obtained from the login route.
/messages/
    WEBSOCKET
        Immediately after connection, an 'authenticate' message must be emitted to the websocket before any other message gets sent in order to continue in the protocol. See the following section for the various protocols in the client:client and client:server interactions.

Messenger Protocol



Git Tips

To setup an ssh-connected server repository, you first want to create your git repository. Follow these steps:

1. SSH to your server. For this tutorial, it is required that the SSH protocol logins solely by key validation and not password. See **Setup Server SSH** if more information is needed.
2. Create or change to directory where repository files will be dumped.
    > mkdir /myServerProject/
    > cd /myServerProject/
3. Assuming git is installed, use the following command:
    > git init --bare
    * A bare git repository doesn't collapse all git files into a hidden directory named '.git, instead it dumps all git helper files the repo directory.
4. Now on the machine that will make commits and be pushing to this server, in a terminal, change directory to your local machine repository.
    > cd /repositories/myServerProject/
5. Call the following command to make a new remote:
    > git remote add myProductionRemote user@0.0.0.0:/myServerProject/
6. For this tutorial, the goal is to have the remote server have its filed updated whenever the GitHub origin remote is committed to. Navigate to this directory:

    /repositories/myServerProject/.git/hooks/

    Create a file named 'post-commit', or if it exists, open it with a text editor. If you had to create this file, it is necessary to give it executable permissions. Edit this file and at the end of its contents add a new line:

```
git push myProductionRemote master
```

Swift Development Tip

• External libraries for our MacOSX Swift client were installed and integrated using the pods system. This requires a file named '*Podfile*' to be created in the directory of your applications main repository. Once you've created (or whenever updating) your podfile, call the command in a terminal on the directory of your *Podfile*:

```
pod install
```

Note: It may be necessary to append ' --repo-update' to the end of the previous terminal command if any pods are out of date.
If you are using an Xcode project with libraries installed by the pod system, you can no longer use the *.xcodeproj* file to open your project in Xcode. The file to be used is in the same directory as the *Podfile* and has the *.xcworkspace* extension.
When opening the *.xcworkspace* file, you should see your project and another project named 'Pods' open in same development window. If you expand the Pods project and look in the Pods directory, there should be a list of

• Disable secure connection checks for URL sessions by adding to a project's Info.plist when developing locally. Note: Open the Info.plist file as source code instead of pro

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
    <key>NSExceptionDomains</key>
    <dict>
        <key>example.com</key>
        <dict>
            <key>NSExceptionAllowsInsecureHTTPLoads</key>
            <true/>
            <key>NSIncludesSubdomains</key>
            <true/>
        </dict>
    </dict>
</dict>
```

TLS/SSL

The server uses LetsEncrypt to provide validate SSL certificates. Since our server has Nginx serving as a reverse proxy (see **Reverse Proxy Setup**) for our NodeJs server application, it's Nginx that handles SSL and not NodeJs. Nginx interacts with

LetsEncrypt's certs to ensure up to TLS1.2 protocol for a secure HTTPS connection. It also handles redirects from HTTP to HTTPS to force all calls to go through TLS. This seems to have negligible overhead for our small-scale system. We decided to handle SSL through Nginx and not NodeJs for better organization and less code in our Node app.

Websockets

Our MacOSX Swift client uses the library *Starscream* for socket connections and our NodeJs server uses its own Net library for socket connections.

The socket stream itself must be validated independently of the web application. So we are using *socketio-jwt* to do that. See this for reference:

> https://auth0.com/blog/auth-with-socket-io/

Nginx

Nginx directs all '/messages' traffic to the node server since node listens on port 8080 and 10001. Port 10001 is for websocket connections. Note that the nginx default webpage is located at:

> /var/www/html/default/index.nginx-debian.html

Reverse Proxy Setup

> Nginx receives all traffic but we want to write our logic in Node, so we direct specific traffic to the node process running through PM2 on the server. In our case, our node server is open on port 8080 (and others), so we redirect traffic to the localhost ip and the port open for our node process. In the example below, all traffic gets directed to the node process, but specific routes can be specified as a location

```
server {
      # Directs traffic at '/' route to a local ip/port.
      location / {
            proxy_pass http://127.0.0.1:8080;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
       }

       . . .
```

```
}
```

Changing Config File

The Nginx configuration file has the file location on Ubuntu 14:

```
/etc/nginx/sites-available/default
```

After making changes to this file, you can error check it before restarting nginx with the following command:

```
sudo nginx -c /etc/nginx/nginx.conf -t
```

After correcting errors, restart nginx for the changes to take affect with the following command:

```
sudo service nginx restart
```

Logging Access

Our Nginx is setup with a custom logging format and location using the following setup:

```
log_format compression '$remote_addr - $remote_user [$time_local] '
        '"$request" $status $body_bytes_sent '
        '"$http_referer" "$http_user_agent" '
         '"$gzip_ratio"';


server {

    gzip on;
    access_log /var/log/nginx/nginx-access.log compression;

    . . .
}
```

This will put connection logs into the following file location:

```
/var/log/nginx/nginx-acces.log
```

Encrypting with Apple Libraries

Encrypting

Apple's Security library is a robust API for performing various security operations. Encrypting information happens through the following call:

```
SecKeyCreateEncryptedData(_:_:_:_:)
```

If the algorithm specified is AES, for example, you would still provide a private RSA key but the library would: generate an AES key, encrypt the data with the AES key, then encrypt the AES key with the RSA key and package this together as data that can be sent over a network.

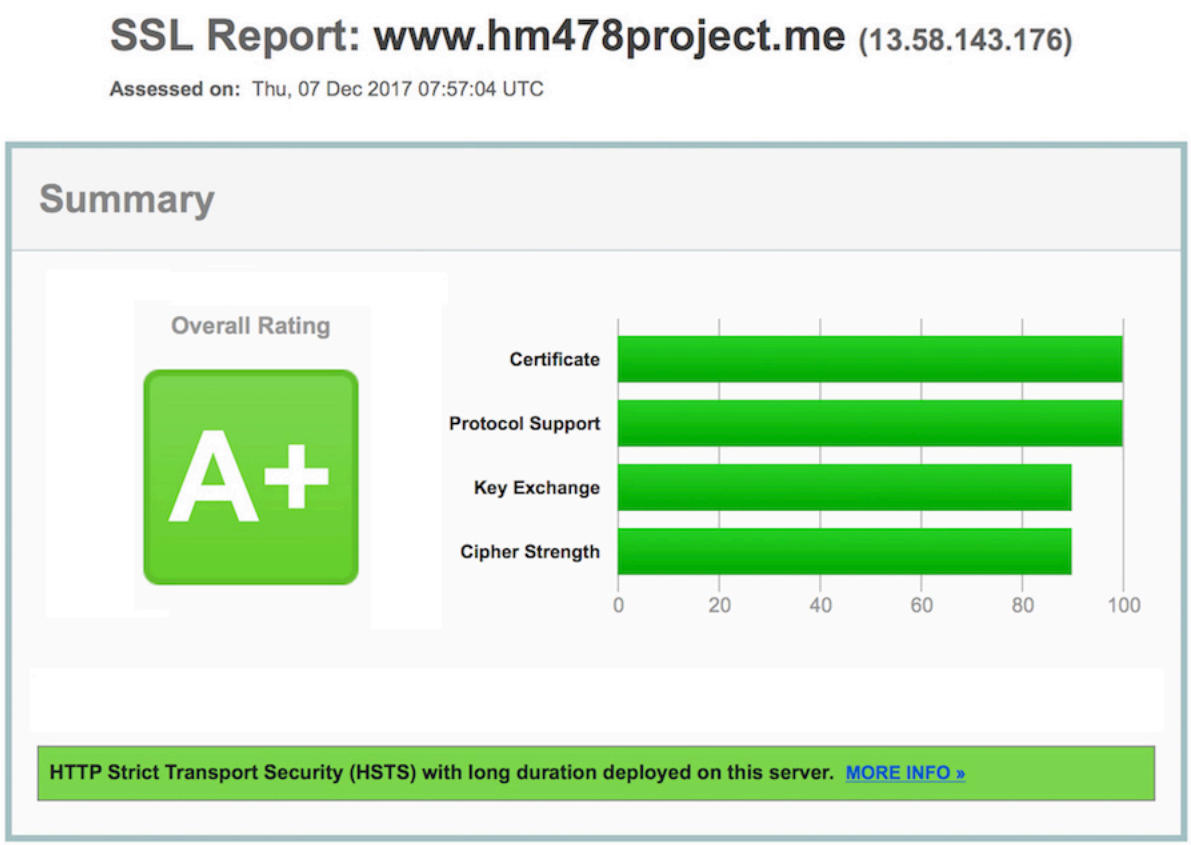
From Apple Security framework documentation:

> On the encryption side, instead of simply padding and encoding the given block of data with the key, the SecKeyCreateEncryptedData(_:_:_:_:) function first generates a random Advanced Encryption Standard (AES) session key. It uses this key to encrypt the input data, and then RSA encrypts the AES key using the input public key you provide. It finally assembles the RSA encrypted session key, the AES encrypted data, and a 16-byte AES-GCM tag into a block of data that it returns to you.

Security Layers

SSL

All connections to the server are directed through the HTTPS protocol. Our SSL certificate is distributed through LetsEncrypt and generated through LetsEncrypt's CertBot. *SSLLabs.com* gives our server the following grade...



Registration

When a new user registers with a unique username, the password is stored in an unrecoverable hash form (SHA256). The password is salted with 128 bits of CSPRNG integers.

Login

After registration, user passwords are never sent to the client. The server uses a double login api route (*login1* and *login2*) in which a challenge is generated and HMAC's generated from the client and server are compared to ensure the password entered on the client matches the hash on the server. If this remote login protocol is successful, the server returns a JWT to the client.

JWT

Since chat operations happen over a websocket, it's necessary to ensure the security of the open socket stream. When a websocket is opened from a client to the server, the first message sent to the server must be a valid JWT. This prevents the stream from getting highjacked as any new connections must provide a valid JWT.

End-to-End Encryption

To guarantee that the server cannot read user chat messages, 2048-bit RSA keys are exchanged between the two individuals in a chat. Both public keys are exchanged after JWT websocket stream validation. These keys are used by the Apple Security libraries to encrypt user messages before they are sent to the server. The encryption process involves taking plaintext and performing a GCM mode AES encryption. The symmetric key generated for this encryption is then itself encrypted by the 2048-bit RSA public key with PKCS7 padded OAEP-RSA and then packaged with the cipher text. See the section **Encrypting with Apple Libraries** above for more information. Messages are not archived and only pass through server memory. Since the server uses websockets and not polling, it isn't necessary to store messages in a database.