SAFE Technical Document

Version 1.0

Capstone Team Winter & Spring '23

Authors

Leshi Chen

Alex Harris

Development Team

Team Lead: Leshi Chen

Dylan C. Conklin

Jeff McHale

Duc Minh Ma

Alex Harris

Vincent Liang

Jaafar Rodgers

Ricardo Sanchez

Table of Contents

Forward - A Message From The Spring ‘23 Team

First, congratulations on getting the SAFE project! We were very excited as a team to be the foundational group to kickstart this into something meaningful that can be passed on from team to team. We think you will enjoy your time with this project as there isn’t just one specialization - you have frontend, backend, script management, and database systems to work with. This project is very fruitful and has a place for everyone to work on something.

We not only dedicated ourselves to writing clean meaningful code and comments, we spent a lot of time refining and revising this document to be concise and informative to the necessary extent without breaking down every line of code in excruciating detail. You will find plenty of information regarding what code is actually doing in the comments, but this is a more detailed breakdown of the code than what you will get out of comments.

Please, take the time to read everything in this document as it could save you potentially hours of troubleshooting - we promise.

Between this document (and the various readme files you’ll find on the branches) there’s enough information - we believe - to catch a team all the way up and have you hit the ground running.

Additionally, leave time at the end of your project to update all the documentation. There’s a lot that changes from team to team and having documentation that is sufficient and clear is something that can make or break a team's success in the long run.

You’ve made it this far, there’s only a little bit more to go!

Code Base Structure

The code base is a combination of TypeScript, JavaScript, Python, raw HTML and CSS encapsulated into a React app that was generated with the command `npx create-react-app my-app --template typescript`. All of this interfaces with a PostgreSQL database hosted by the CAT to store the information provided from the Sender at *feedback.cs.pdx.edu* through the use of a REST API script created using Express, where all of this lives on the virtual machine setup by the CAT at *feedback.cs.pdx.edu*. At the start of this project the team elected to use TypeScript due to its strong type enforcement and the ability to interface nicely with a vast number API available within the Node.js environment with support through the Node Package Manager (npm).

Anecdotal but worth mentioning, even the Node.js founder has stated people should be using TypeScript over JavaScript and gave praise to the team at Microsoft that is responsible for maintaining it. Understanding how the code is structured and how it interfaces all with each other is the hardest part to overcome when looking at a project that's already established. We as the founding team hope to facilitate a certain level of understanding that allows the following team (or teams) to understand how everything works. This section will not cover how specific files or scripts work but how you can work with the base tools available to you with React applications.

Additionally, there are supporting comments in most of the files that help explain either what is happening in the file, or that specific function.

Node.js

The overarching base to understand the functionalities of is Node itself and its package manager, npm. Node allows us the ability to interface with the JavaScript runtime environment in a system agnostic way - much like Java! This allows for the SAFE application to be loaded up and started nearly anywhere with minimal setup. Node also allows us to run the REST API (discussed later) in the background with PM2 (more on this in the "How can I start the REST API script to receive feedback?" section) while the website is running and waiting for a user to engage with it without having to actively be logged into the virtual machine.

npm

While npm allows for ease of use to install and manage various packages, including updating them. Due to the constraints of the CAT having to manage the virtual machine that hosts this content. You will find that npm will alert an individual quite often if there's an update to npm, or that there are vulnerabilities detected and you should update to fix these issues with `npm audi fix --force`. What we can say is that you needn't worry about this unless you are using one of the packages with vulnerabilities, which, at the time of this document's creation, is only `nth-check` and exists globally, which must be corrected by the CAT, **however** fixing it breaks a number of other packages and most importantly the React-scripts package. Avoid the `nth-check` package until this issue has been resolved at the source level within npm.

Some commands we used - or are using in the python script - from npm are…

- `npx tsc` to compile the necessary files to support and create the `server.js` to act as the REST API.
- `npm run start` is a React script to emulate the compiled React app and allow updates to the frontend (user interface) live so that a developer can see the changes (or errors) to the website in real time.
  - This option also works on the PSU servers, but there are limitations that may impact your ability to test new features being added to the UI.
- `npm i <package name>` without angle brackets to install a specific package that may be used in the final product and can be imported into any .ts or .tsx file to be used.
- `npx depcheck` to identify any packages that are not being used in the project.
  - Some of these will still alert they are not being used (i.e. `@types/jest`) which is true but **are part of the testing suite and should not be removed from the package.json**.
- From the SAFE directory - `node JSoutFile/safeMessageDB/server.js` will run the REST API script in the background listening on a port set within the script.
- `npm run build` is a React script that does the heavy lifting of taking the source located in the `src` folder and compiles it from TypeScript into JavaScript and performs the extra additive steps to the necessary html files.
  - We encourage the next team to examine the before and after of the `index.html` so that you can see the minor differences in the files.

package.json and package-lock.json

In `package.json`, you will find the properties that apply specifically to npm and node. This includes packages, React scripts, and some additional requirements. Take note that in this file exist comments too but must exist as properties of the file because json does not support traditional commenting as it's not a language but a data-interchange format.

There are two fairly unique items in the "scripts" section of this file. One specifically for Portland State University deployment and another that is for debugging purposes. This second one acts as a set of logical statements such that the proper python command is called depending on the system. For one reason or another, on Unix-like systems it is python3 but on Windows it is python. This npm script is here for a developer to use if they wish for full details while running various stages of the script or to execute on Windows machines. More details on the script are located in the "safe_setup.py Script" section of this document.

In `package-lock.json` exists to record the specific versions of packages, their dependencies, and sub-dependencies. The purpose for this file is to ensure that the same exact version of each package is used across the development environment. This file has been added to the `.gitignore` due the fact that it is not necessary with this project's lack of complexity and restrictive requirements.

tsconfig.json

This file represents the overall behavior of the TypeScript (TS) compiler. While json files do not support comments natively, you can still use them (as we did) and make note of important things to consider, depending on the file ESLint may get upset saying that commenting is not allowed but we can sneak around this alert in this file. When `npx tsc` is run, this is calling specifically the TS compiler to build the REST API script we run. This is designated in the "include" section in this file and any other required files to be built and run with node independently of the web page itself, again, like the REST API script.

However, this does not apply to `npm run build`, which is a React script.

.ts files

These files are pure TypeScript. These files contain no React components and require/implement no JSX. What these files do contain are typical variables, functions and/or classes that are used to support the backend of the system, or perform minor operations to support the JSX that may exist in the .tsx files.

.tsx files

As previously mentioned, these files contain JSX code which produces React elements. What you'll see in these files are the components being used to generate the frontend and provide a usable experience to the users. These distinctions are important to maintain to ensure that the project keeps a clean structure.

However, with TypeScript 1.6 this distinction is no longer of critical importance, but many projects carry the same patterns and practices forward because of the clarity in the distinction.

How does the website generate the components?

Within the base directory (`./SAFE`) and the `src` folder, you will notice there are a number of folders and files. The key ones you will want to look at are the `public` folder and the `index.tsx` as well as the `index.css`. Take a moment and read the comments in these files. Firstly, within `index.tsx` you'll notice that a `const` object called `root` generates *all* of the webpage by wrapping the id `root` and casting it to the type `as HTMLElement` (the as after and the type following is *just one way* to cast in TypeScript with JSX.)

Next the `root` object will call `render` in strict mode which provides notice if any errors are present. In this it will create a new JSX element that is wrapped into the `index.html` body. Since the UI is in a modularized design, this effectively cascades down and renders all the elements you'll be able to find in the safeUI folder, all of which are designed using Material UI (more on this later in the "safeUI Folder" section, but to sum it up it's a lot like Bootstrap but for React.)

All of the elements are rendered instantly, and the user on the frontend will not even notice it happened. This approach is much more seamless than what you'll find on websites exclusively written in pure HTML and CSS.

Lastly, it is recommended reading the setup instructions located in the GitHub repository for either the `main` or `MVP`. More about the requirements on how to set up the webpage for both the virtual machine and on your own accounts within the MCECS network are found throughout this document. However, the easiest approach will be to visit these two branches as they have different `README.md` files each.

Where are the classes?

A lot of TypeScript projects utilize classes, but as you may have noticed there are no classes in *this* project. We had initially started with classes but found that we didn't need them because the project was far less complex than we had initially thought. We discovered this fairly early on, so it wasn't an issue for the length of the project. Our initial thinking was that they could be used as vehicles to contain and transport data between core components of the project, but again, that was deemed unnecessary early enough we could adjust with ease to proceed without them.

How can I access a PSQL database on the web?

From a web browser of your choice:

1. Go to https://dbase.cecs.pdx.edu/postgresql/
2. Enter the provided MCECS PSQL Username for the database for both "username" and "database."
3. Enter the password for the database.
4. Then simply login

What you will notice for the production database is that there are three tables, an Admin, Message, and MessageTest. Due to technical limitations, MessageTest must be used for writing tests while developing to have successful tests. Do not run tests against the production table! Incredibly dangerous and can cause significant issues.

How can I start the REST API script to receive feedback?

There are a few situations where you may need to do this:

1. **From a PSU server such as ada, babbage, rita or quizor.**
    a. Navigate to `~./public_html/SAFE`.
    b. On your command line run the function `npx tsc` to compile the `SAFE/safeMessageDB/server.ts` to a `.js` file.
    c. Execute `node JSoutFile/safeMessageDB/server.js`
        i. If this succeeds, it will notify you it is listening on a specific port
        ii. If this fails, it will error out. This usually means that someone else is using that port currently and you need to modify your port, recompile the server code, then execute the command again and it should work.

2. **From the *feedback.cs.pdx.edu* virtual machine. This is a bit more complicated.**
    a. Navigate to `~./public_html/SAFE`.
    b. Type `pm2 list`.
        i. If it is running type `pm2 delete SAFE_SERVER`.
    c. Then type:

    ```
    pm2 start SAFE/JSoutFile/safeMessageDB/server.js --name
    SAFE_SERVER --watch "safeConfig/safeConfig.json"
    ```

    d. From here, confirmation should be stated in the terminal that PM2 started the process.
    e. To verify this, type `pm2  list` again and you should see PM2 is running the script.
    f. The `--watch` option followed by a directory AND file tells PM2 that if this file *ever* changes, restart the script.
        i. This is done to support any changes to the `safeConfig.json` file that might happen. The REST API script starts with fixed information, so if that information changes then the script must restart to capture those new changes.

NOTE: if the REST API code is ever changed/recompiled, the PM2 process that is being run must be fully restarted. **You must delete it and then run the start command mentioned in 2.c.**

3. **From your own personal machine, locally, and no SSH.**
    a. You can’t. The reason being is that the webpage (even when being emulated by the start React script) is not able to communicate with the PSU network.
    b. Because of this restriction you also cannot perform tests from this setup.

Future capstone teams may want to add this capability to the python script for ease of use.

Extensions

Throughout the development process, the team discovered several extensions that became regularly used tools. Some of these may look familiar immediately, and others less so. If you're not certain about a particular extension, it is encouraged that you read what the extension does before writing it off. Below is the list of extensions the team suggests future development teams use.

- Code Spell Checker
- Debugger for Firefox
- Debugger for Chrome
- ES7+ React/Redux/React-Native snippets
- ESLint
- html to JSX
- Import Cost
- Live Server
- Jest
- Markdown All in One
- Material Icon Theme
- npm Intellisense
- Prettier - Code formatter
- Simple React Snippets
- Todo Tree
- VSCode React Refactor
- GitLens (No need to make an account!)
- Black Formatter (Python formatting tool)
- Pylance
- Python Extension Pack

Minimum Viable Product (MVP)

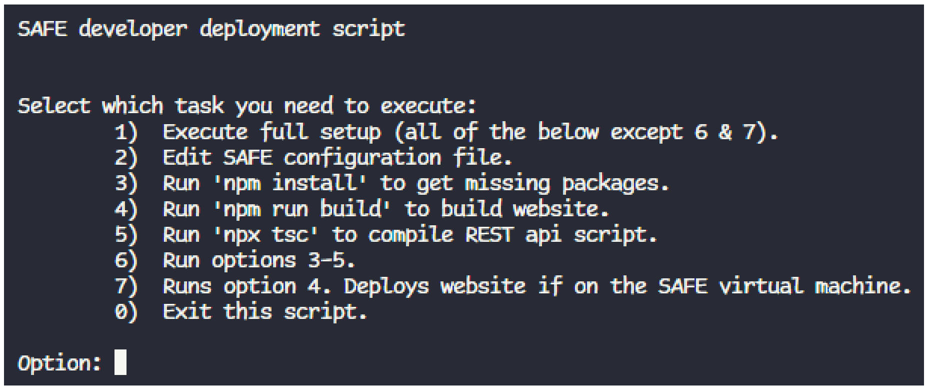
This section aims to provide developers with a comprehensive understanding of the implementation files in the MVP branch. It includes detailed descriptions of each file, the packages and libraries utilized, as well as an explanation of how the code accomplishes common tasks and functionality. Moreover, it offers a thorough explanation of the parameters and potential error conditions developers may encounter, particularly when working with the back-end code. Additionally, the section covers the logic and algorithms employed in the MVP codebase.

safe_setup.py Script

This python script allows devs or owners to quickly setup and/or rebuild all/part of the code to be deployed without having to sweat overhead work of ensuring they are clearing the correct files or setting up the system from the start correctly.

Features

- Menu to enable various actions available to the user.



```
SAFE developer deployment script


Select which task you need to execute:
        1)  Execute full setup (all of the below except 6 & 7).
        2)  Edit SAFE configuration file.
        3)  Run 'npm install' to get missing packages.
        4)  Run 'npm run build' to build website.
        5)  Run 'npx tsc' to compile REST api script.
        6)  Run options 3-5.
        7)  Runs option 4. Deploys website if on the SAFE virtual machine.
        0)  Exit this script.

Option:
```

  - To execute this on *most* PSU systems (ada, babbage, rita, quizor, and the virtual machine) it is recommend that you run the npm script `npm run psu_deploy`
  - **Option 1** always runs automatically if the script detects that it is missing `safeConfig.json`. This option first generates the configuration file, builds and then sets necessary permissions for specific files and folders to be visible by the PSU apache server. It executes every required step on initial setup that enables the entire website to come on line for the CECS/MCECS PSU web servers.

- Other options enable real time rebuilding while editing code to support live development without npm run start by targeting specific key sources to be rebuilt if changes are made.
- **Option 7** will only successfully and fully execute if it is being run on the VM `feedback.cs.pdx.edu`, otherwise an error pops up alerting the user of this fact.
- This script can sit and be on standby in the background until a developer is ready to use it, or can be run at any point to execute the laid out tasks in the screenshot above.

- This script is written to execute around two specific systems: Windows-like and Posix-like (Linux, Mac, etc.)
  - For Posix-like systems, `chmod` will be executed and establish correct user group permissions based on the current `hostname`.
  - If the system is a Windows-like system, `chmod` is never called. `chmod` is only ever called on Posix-like systems.
    - **On Windows-Like systems, the script will ONLY execute if it is ran in debug mode.**
  - Other types of operating systems have not been accounted for.
- To run the script in **debug mode** it is recommended to run it with the npm command `npm run debug_deploy`. This will seamlessly and safely perform this action for you regardless of which of the two valid systems you run it on.
- This script also checks for various `hostname`s to ensure that it's not accidentally run where it shouldn't be.
  - A regex examines the system's hostname; if it isn't a part of the regex, the script returns a message. Otherwise it proceeds as expected.
- Why does this script care about which system or `hostname` it is ran on?
  - To answer the first part, most of the function calls in the script will not work on one but will on the other, and so user safe-guards are in place to ensure they do not slip into a line of code whereby a stack-trace is produced.
  - The `hostname` protects the user, again, from executing commands and functions in such a way that they do not cause the script to fail.

    - The script is designed to only support the two most popular systems actively used today - Posix-like and Windows-like. This is to ensure that any common developer can utilize this without having issues.

safeMessageDB Folder

---

server.ts

This is the server code responsible for facilitating data transfer between the front end and back end and acts as the REST API for the system. The server supports various request methods, such as GET, POST, PUT and DELETE, to handle data operations. It provides endpoints that are necessary for the front-end code to function properly.

This file *must* be compiled to JavaScript with the command `npx tsc` any time the contents of the file changes. We recommend using the python script to perform this operation.

- The server is using the Express.js framework with the following line of code:

  ```
  const app = express();
  ```

- The server is using the CORS middleware to allow cross-origin requests to the Express.js application from any domain, with the following line of code: `app.use(cors());`
    - However, this approach is only recommended for development environments. In production, you should change each endpoint's CORS settings to the specific URL.
- For reference, the ip address you'll see below - `131.252.208.28` - is the ip address of `rita.cecs.pdx.edu`.
    - This is the server where we ran most of the development until the VM came online.
    - **Future capstone teams can inquire with the CAT about using/establishing users on the SAFE VM for development purposes.**

```
// Define an endpoint for adding a new event
// This is a POST request to post data into the database
app.post('/addMessage', async (req: Request, res: Response) => {
  // Get all data from the request body and store them into these object
  const {. . .} = req.body; // Vars to store info from the form
  try {
    // Acquire a client connection from the connection pool
    // it will return the db.pool from the connect() function
    const client = await messageDBConnect.connect();
```

```
    // Execute a SQL query to insert a new event
    await client.query(. . .);
    // Release the client connection back to the pool
    await client.release();

    // Send notification email to receiver
    const mailArgs = [
      `-s "[SAFE FEEDBACK] - (${sanitizedTitle})"`,
      getConfigProp(sjp.rcvr_email, scp), //get email address from Config
    ];
    //send out email using spawn to create a child process in terminal
    const mail = spawn('mail', mailArgs);
    mail.stdin.write(. . .);
    mail.stdin.end();

    //status 200 to indicate success.
    res.status(200).send();
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Above is the code for the POST request. The server has an endpoint `/addMessage` that is used to insert messages into the database. It first gets all the data from the request URL body and stores them in different objects. Then, it sanitizes the body to prevent XSS attacks using the XSS package. After that, it replaces anything that is not a letter, number, or '/' from the title. The server then tries to connect to the database and makes a query to insert the message based on the data we got from the URL body. Finally, it sends out an email using the spawn package and writes the title and message body in the email.

The following code snippet is used to make the server listen to the IP address' port:

```
const port = 3001;
app.listen(port, '131.252.208.28', () => {
  console.log(`Server listening on port ${port}`);
});
```

In the production environment in the VM, here is what you need to use:

```
// Start the server
app.listen(3001, '127.0.0.1', () => {
  console.log(`Server listening on port 3001`);
});
```

It is being bound to the localhost `127.0.0.1` because the CAT has setup a proxy of `https://feedback.cs.pdx.edu/addMessage` to `localhost:3001`, hence we can fetch the request with https and leave the SSL requirement behind.

While on `rita`, when using the fetch API on the UI, the following code snippet can be used to send a request to the server's endpoint, catch the response, and identify any errors from the endpoint:

```
const port = 3001;
fetch(`http://131.252.208.28:${port}/addMessage`, {
 method: 'POST',
 headers: {'Content-Type': 'application/json'},})
 .then((response) => {
   if (!response.ok) {throw new Error('Network response was not ok');}
   else {setOpenSuccess(true);}})
 .catch((error) => {
   console.error('There was a problem with the fetch operation:', error);
   handleOpenError();
  });
```

In the production environment in the VM (*feedback.cs.pdx.edu*), here is what you need:

```
const port = 3001;
fetch(`https://feedback.cs.pdx.edu/addMessage`, {
 method: 'POST',
 headers: {'Content-Type': 'application/json'},})
 .then((response) => {
   if (!response.ok) {throw new Error('Network response was not ok');}
   else {setOpenSuccess(true);}})
 .catch((error) => {
   console.error('There was a problem with the fetch operation:', error);
   handleOpenError();
  });
```

NOTE: While these two fetch requests look incredibly similar, the beginning of the fetch is **incredibly different**.

The reason this fetch is formatted this way is because CAT sets a proxy on the addMessage URL. And now we only have the addMessage proxy available. If you need to add more endpoints to fetch from it, CONTACT CAT.

For **testing**, basically create some testEndPoint in the server, and then modify the database table name to ‘MessageTest’ we had created.

messageDBConnect.ts

- This should be imported whenever there is a need to connect to the database. It exports the `messageDBConnect` as a new Pool object from pg which can later be used to invoke `.connect` and retrieve a `pg` client for initiating database operation.
- The database login information is concealed behind the scenes, and it utilizes the `getConfigProp(prop: String, path: String);` function from the `Util` class to retrieve the required information.

safeUtil Folder

Util.ts

This file contains functions that can be imported into any other file and used to perform actions. With this version of the document (and the system) it contains very few available actions, but these are powerful and incredibly useful tools used throughout.

- `safeJSONProps` is an enum that contains a list of specific `const` strings to be used and passed to `getConfigProp`.
- `getConfigProp` takes a `prop: String` and a `path: String` both of which live inside this file too.
  - `prop: String` comes from the enum `safeJSONProps`. This enum contains strings that are required to be used when requesting specific properties from the `safeConfig.json`.
  - `path: String` comes from the `const safeConfigPath` which is exported by this file to be used elsewhere in the system such as the `server.js` file.
  - **This is done this way such that no conflicts can arise from attempting to access a value that doesn't exist.**
- `safeConfig.json` is generated by `safe_setup.py`. This contains an explicit list of JSON properties that the system is required to use to both connect to the database and email the Receiver(s) on file.
- `const safeConfigPath` is an absolute path to where `safeConfig.json` lives.

safeUI Folder

SafeUI.tsx

This file represents the frontend code for the Sender-side user interface. The majority of the components were written with MaterialUI. Typescript was used, although some of its benefits were not utilized. One of the most important aspects is that of defining interfaces to ensure type safety (potential use cases discussed in submitForm.tsx).

SafeUI.tsx merely wraps modularized components defined in the Component folder. This will hopefully improve the overall system readability and maintainability as SAFE continues to be developed and passed from capstone team to capstone team.

Components Folder

This folder contains four .tsx files and an additional Form Components folder.

- `bannerBar.tsx`
  - Basic function that returns a green banner with PSU logo. No functionality is built into the logo as of now. Note that the transparent logo used must be placed behind 'PSU Green' `(#6a7f10)`
- `headerText.tsx`
  - Basic function that displays website title text. Text responsiveness is tied to static layout currently. Consider replacing it with media queries for more dynamic execution. File also contains a link to the 'about.html' page, explaining how this system protects its users identity.
- `submitForm.tsx`
  - The most dynamic file in the UI folder. This component renders the form UI and processes the submission data. It uses various hook statements and event handler functions that manage and update the current state of a feedback message.

    - This is where interfaces would prove useful. Instead of defining numerous hooks to set an updated value, we could work with a Feedback object that updates these values all at once. We could manage the state more effectively, overall limiting the behavior and improving readability. Within the handleSubmit handler function, we could simply pass a Feedback object instead of specifying all individual fields.
    - SubmitForm is made up of 7 different 'micro' components, found in the Form Components folder. These will be discussed briefly further down.
- `eventHandler.tsx`
    - Modular handler functions that were able to be pulled from submitForm.tsx to reduce code build up. All handlers could exist in one file with an interface defined.

Form Components

There are 7 different .tsx files that make up the Form Components folder. This comprised the bulk of our code and we figured it best to break each into their own component to be independently rendered in submitForm.tsx.

- `toField.tsx`, `subjectField.tsx` and `messageField.tsx`
    - These files make up the actual form component. ToField() is a static value, but you could easily update this to receive input and limit character input. SubjectField() is capped at 100 characters and MessageField() is capped at 7500 (~2000 words). Maybe a more dynamic word count approach would be better served here.
- `submitError.tsx` and `submitSuccess.tsx`
    - These files use MUI dialog 'pop out' components to convey if a Sender's submission was successful or not. An improvement would be to combine these into one file. There are no tests related to these components. If further work is done to the safeUI, our suggestion is to add tests to these components.
- `submitButton.tsx`
    - File for the button component. References StyledSubmitButton which is found in the Styles folder.
- `titleNine.tsx`
    - Used to inform Senders that this feedback system should not be used to report Title 9 concerns. Includes links to potentially appropriate resources for the user.

Public Folder

Index.html & Index.css

These files are the main website that generate the components from the files located in `src` after being compiled from TypeScript to JavaScript with `npm run build`. The build script does not include the REST API as that is independent of the user interface.

The CSS file offers some basic stylizing for the HTML file. The HTML file is, in all honesty, a container for the JavaScript that is produced by the compilation process with the build script, and doesn't feature much outside of serving this purpose. It sounds unimportant, but without it there wouldn't be a landing page for users to visit and interact with.

About.html & About.css

The `about.html` provides visitors with essential information about the website and what its objective is. The about page of this project describes how we are committed to users anonymity to help build trust and establish a connection between the Computer Science department and its students.

The `about.css` is the styling sheet for the About.html and most classes were made with consistency of the whole UI in mind. The span class and all of its nth children are used to create the text animation when the page first loads, but it's not limited to that. The span tag can be used to wrap any element in the page to create the blur animation effect.

Beyond MVP

This section serves a similar purpose as the MVP section, providing developers with insights into the implementation files This section only contains additional features that have not yet been released or deployed to the SAFE domain. Along with descriptions, package and library information, and explanations of common tasks and interactions, this section offers details about the upcoming features. It covers the parameters, potential error conditions, and algorithms associated with these unreleased features in the codebase.

safeMessageDB Folder

---

server.ts

Note that we didn't fully test all these endpoints yet. So some fetch syntax might not be 100% correct.

- `getallmessages` endpoint

```
// Define an endpoint for retrieving all messages with it's all info
app.get('/getallmessages', async (req: Request, res: Response) => {
  try {
    // Acquire a client connection from the connection pool
    const client = await messageDBConnect.connect();

    // Execute a SQL query to retrieve all messages
    const result = await client.query('SELECT * FROM "Message";');
    //if no message return
    if (result.rows.length === 0) {
      res.status(404).json({ error: 'No message in database' });
    } else {
      //return all the message getting out
      res.status(200).json(result.rows);
    }
    // Release the client connection back to the pool
    client.release();
  } catch (err) {
    console.error(err);
    res
      .status(500)
      .json({ error: 'Internal server error, please try back later' });
  }
});
```

- `deletemessage` endpoint

```
// Define an endpoint for deleting a message
app.delete('/deletemessage', async (req: Request, res: Response) => {
  const { code } = req.query;

  try {
    // Acquire a client connection from the connection pool
    const client = await messageDBConnect.connect();

    // Execute a SQL query to retrieve a particular message based on code
    const result = await client.query(
      'DELETE FROM "Message" WHERE code = $1;',
      [code]
    );
    // If can't find message with corresponding code
    if (result.rowCount === 0) {
      res
        .status(404)
        .json({ error: 'No matching record found with provided code' });
    } else {
      // Send the message back to front-end
      res.status(200).json('Message deleted!');
    }
    // Release the client connection back to the pool
    client.release();
  } catch (err) {
    console.error(err);
    res
      .status(500)
      .json({ error: 'Internal server error, please try back later' });
  }
});
```

- `getmessage` endpoint

```
// Define an endpoint for retrieving one message
app.get('/getmessage', async (req: Request, res: Response) => {
  const { code } = req.query;

  try {
    // Acquire a client connection from the connection pool
    const client = await messageDBConnect.connect();

    // Execute a SQL query to retrieve one message
    const result = await client.query(
      'SELECT message, message_reply FROM "Message" WHERE code = $1;',
      [code]
    );
    // If can't find message with corresponding code
    if (result.rows.length === 0) {
      res
        .status(404)
        .json({ error: 'No matching record found with provided code' });
    } else {
      // Send the message back to front-end
      res.status(200).json(result.rows[0]);
    }
    // Release the client connection back to the pool
    client.release();
  } catch (err) {
    console.error(err);
    res
      .status(500)
      .json({ error: 'Internal server error, please try back later' });
  }
});
```

- `addReply` endpoint

```
// Define an endpoint for adding reply to the message
app.post('/addReply', async (req: Request, res: Response) => {
  const { code, reply } = req.body;

  try {
    // Acquire a client connection from the connection pool
    const client = await messageDBConnect.connect();
    // Execute a SQL query to insert a new message
    await client.query(
      'UPDATE "Message" SET message_reply = $1 WHERE code = $2;',
      [reply, code]
    );
    // Release the client connection back to the pool
    await client.release();

    res.status(200); //we use this to test if user can get back the code from server
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

- `setReply` endpoint

```
// Define an endpoint for setting receive_reply to true
app.post('/setReply', async (req: Request, res: Response) => {
  const { code } = req.body;

  try {
    // Acquire a client connection from the connection pool
    const client = await messageDBConnect.connect();
    // Execute a SQL query to insert a new message
    await client.query(
      'UPDATE "Message" SET receive_reply = true WHERE code = $1',
      [code]
    );
    // Release the client connection back to the pool
    await client.release();

    res.status(200); //we use this to test if user can get back the code from server
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

- `receiverEmail` endpoint

```
// Define an endpoint for emailing code to Sender
app.post('/receiverEmail', async (req: Request, res: Response) => {
  const { email, code } = req.body;

  try {
    // Send notification email to Sender
    const mailArgs = [`-s SAFE- This is a copy of your Code`, email];
    const mail = spawn('mail', mailArgs);
    mail.stdin.write(
      `Here is a copy of your unqiue code: ${code} \n please check back later in the website
with reply. https://feedback.cs.pdx.edu`
    );
    mail.stdin.end();

     res.status(200).send(`Your code had been sent to the email you provied`); //we use this to
test if user can get back the code from server
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

verifyString.ts

`Function checkString()`

This function verifies that a string is not empty, doesn't contain curse words, and performs sentiment analysis on the string. It will throw an error if the string is empty or contains a curse word, and return the comparative sentiment score.

The curse word filtering is done using the bad-words package.

The sentiment analysis package has two scoring systems: a normal score, and a comparative score. Each word in the string is analyzed by the sentiment analysis package, with certain words having a score based on connotation and context. Words with positive connotations have positive numbers, and words with negative connotations have negative numbers. All other words are neutral. The score for each word may be higher or lower, depending on how extreme the word is.

The regular score is calculated by adding up all the scores of words with positive connotations, and subtracting the scores of all the words with negative connotations. The result is an integer, with a positive number indicating an overall positive review, and a negative number indicating an overall negative interview.

The comparative score is calculated by taking the regular score (the difference of the sum of all positive word scores and the sum of all negative word scores) and dividing it by the total number of words in the string. We chose to use this score, because we feel it most accurately represents the sentiment of reviews given to the feedback system. A user could theoretically leave a long review, and have a low negative score due to the length of the review, while another review could have a negative score closer to zero, and actually have expressed a worse experience with fewer words. Using the comparative score could balance out these reviews, and more accurately calculate who had a worse experience, regardless of the length of the review.

safeUtil Folder

---

generateCode.ts

There are two variables that refer to the same thing. A code combination is an array of integers corresponding to values in the dictionary. The code is the string that is generated using the code combination and the dictionary. The initial code combination is set to an array of random integers. To keep things simple, only the code combination is passed between functions. The code rendered by the code combination, and the dictionary is only used to query the database, and return the available code through `genCode()`.

This file generates a unique code, which is the identifier for each message in the database. It starts by generating a random code combination, queries the database to check if the code is available, and generates the next available value, if the initial value was taken.

A note on parameters: each function serves a different purpose in the process of finding a unique code, and uses many of the same variables. For simplicity, these arguments are accepted in the same order, wherever possible.

Helper Functions

- `randInt(max: number, min: number = 0): number`
    - This is a basic random number generator function. You can give it a maximum, and an optional minimum. The two numbers are inclusive (e.g. randInt(3) will produce either 0, 1, 2, or 3, randInt(3, 2) will produce 2 or 3, and randInt(5, 5) will produce 5).
- `renderCode(code_combination: number[], dictionary: string): string`
- `renderCodeCombination(code: string, dictionary: string)`
    - These two functions are inverses of each other. Render code, takes a code combination as input, and produces a code string based on the dictionary. Render code combination does the opposite, taking a code string as input, and produces the code combination corresponding to it, based on the dictionary.

Main Logic

- `genCode()`
    - This function creates a dictionary, which is just a string of valid characters for the code to have, and generates a random code combination based on the dictionary for the code combination to initially have. It then calls getCode() to verify the code is available, and getCode() will change the value if it is not. genCode() then returns the rendered code string corresponding to the selected code combination.
    - This is the only public function in the class, the rest of the functions are internal logic.
- `initCodeCombination(code_length: number, dictionary: string): number[]`
    - This function creates an initial code combination of a certain length, using random values from the dictionary.
- `getCode()`
    - This function cycles through all values of the code combination until it arrives at the final value in the code combination. It then calls `queryCodes()` to check if the code combination is available, and tries other code combinations if similar codes are unavailable.
    - In the event that all similar code combinations are taken (where the final value in the code combination is a wild card), we will then work backwards through the code combination, cycling through all possible values at each position, until a code combination is available. A simpler way to visualize this is to think of binary sequences, when you add one to each sequence. I will show an example below.

Visualization of `getCode()`

- Dictionary = “abc”
- Code combination = [2, 0, 1]

Call #	Value of index	Value of code combination	Action after calling queryCodes	Possible codes returned by queryCodes
1	0	[2, 0, 1]	N/A, calling genCode	
2	1	[2, 0, 1]	N/A, calling genCode	
3	2	[2, 0, 1]	Return code combination if available, continue otherwise	[2, 0, 1] [2, 0, 2] [2, 0, 0]
2	1	[2, 1, 1]	N/A, calling genCode	
4	2	[2, 1, 1]	Return code combination if available, continue otherwise	[2, 1, 1] [2, 1, 2] [2, 1, 0]
2	1	[2, 2, 1]	N/A, calling genCode	
5	2	[2, 2, 1]	Return code combination if available, continue otherwise	[2, 2, 1] [2, 2, 2] [2, 2, 0]
1	0	[0, 0, 1]	N/A, calling genCode	
6	1	[0, 0, 1]	N/A, calling genCode	
7	2	[0, 0, 1]	Return code combination if available, continue otherwise	[0, 0, 1] [0, 0, 2] [0, 0, 0]
Etc.				

- `queryCodes()`
    - This function accepts a code combination to query a database, and attempts to find an available code similar to the code combination passed to the function. It queries the database using the code combination passed to the function, and returns a list of codes that exist in the database. If nothing is returned from the database, we know the code is available. If not, we pass the code combination and the list of existing codes from the database to isAvailable(), and try to find an available code there.
    - We only query the database using the code combination, and have the last value as a wildcard. This is done bypassing all but the last value of the code combination in the query, matching the beginning, part of the codes, and checking the length of the codes in the database match the length of the code combination we are checking for. There are two reasons why we do this.
    - We do not want large queries going to the database, because the database has to return all the data over a network, and that can slow the application down. It could also be beneficial to set up indexes on the database in the future, so the database does not have to query large sets of data as the database grows.
    - We also do not want to overwhelm the network or the database with queries. If one code is not available, and five consecutive valid codes after it are also unavailable, it would take seven queries to the database to verify a valid code. If we query the database with the last code combination value as a wildcard, we can get more results in less queries. This reduces the chance we overwhelm the database or the network, and each query will have a maximum size of the length of the dictionary.
- `isAvailable()`
    - This function checks if the rendered code combination is in the array. If the code is not in the array, then it is a valid code. If the code is in the array, then it is a valid code only if the existing code in the database is older than 10 years old. this allows us to reuse codes that can be reasonably assumed to be out of date. At the University, we assume most students will graduate in five years or less. If the code is 10 years old, we can reasonably assume the student is no longer at the University, or cares about having the code.

safeUI Folder

There are two main additional features added to safeUI. The first feature is a check reply button. If the user has opted to receive replies he can click on the button and a pop up modal will appear. He can input his code and he will receive his original message and a reply if one exists. Currently this feature is not fully implemented. See below for details.

The second feature is after the submit button is hit. If the submission is successful a pop up modal will appear informing the user that their submission was successful and asking if they want a reply. If yes, then another modal will appear with the uniquely generated code. It will also have an input email prompt if the user wants a copy of this code sent to them.

Components

- `codeModal.tsx`
    - This contains the pop up modal giving the user his reply code.
    - Also prompting the user for their email is he wants the code sent to them.
    - The sending of an email functionality is done in handleEmail in emailEventHandler.tsx.
- `emailModal.tsx`
    - This contains the pop up modal confirming the message was sent and asking the Sender if they want to receive replays.
- `emailEventHandler.tsx`
    - Event handler functions for the emailModal and codeModal.
    - Referenced in submitForm.tsx, codeModal.tsx, and emailModal.tsx.
- `checkReply.tsx`
    - This component features a check-reply button that enables two pop-up modals.
    - The first pop-up modal prompts the user to enter a unique code, which is provided, to retrieve the associated message.
    - The second pop-out modal is triggered to display the retrieved message.

- `uniqueCodeModal.tsx`
    - This component includes a pop-up modal for the user to enter a unique code provided.
    - The input code is validated in the handleCodeSubmit() function to retrieve the relevant data from the database.
- `displayMessageModal.tsx`
    - This component includes a pop-up modal to display the message associated with the provided unique code.
    - The corresponding message is determined in the handleCodeSubmit() function after the code is validated.
- `checkReplyEventHandler.tsx`
    - This component features a check-reply button that enables two pop-up modals.
    - These functions are responsible for managing the opening and closing functionality of both modals, as well as handling code validation and retrieving the message from the database.

safeReceiverDashboard Folder

Components

- `MessageCard.tsx`
  - Displays message details using the MUI Card component. Card is naturally responsive and will expand/shrink to message length accordingly. Layout and display are purely static. MessageCard makes use of an interface, IMessageCardProps, to track date, title, and message fields. This could be updated to track further fields, such as if a message has been read or not.
- `bannerNavBar.tsx`
  - Similar to the bannerBar.tsx in safeUI, but includes a drop down menu on the right hand side to logout or login. User profiles and SSO for the dashboard haven't been implemented, so this will need to be updated to support this functionality.
- `messageBox.tsx`
  - Displays messages received from Senders. Each message contains a sentiment analysis icon, subject, date, and delete option. Currently an arbitrary amount of placeholder messages are being generated. When successful calls are made to the database and live data is used, consider how many messages should be initially rendered to the page. Further, how to refresh this data to get additional messages. Should message flow follow an 'infinite scroll' set up, or should a specified amount of messages be rendered per page. (10, 25, 50, etc.)
  - Contains some helpful notes in `fetchMessages()` to get information from the database.
  - Icon or other visual method should be added to messages to indicate read/unread status for a Receiver.
- `messageControl.tsx`
  - Code for the feedback bar spanning the message box and message card components. Contains a drop down menu to filter messages between read, unread and replied to.

safeReceiverDashboard.tsx

Receiver dashboard is rendered here. Just like in safeUI, design for the dashboard was intended to be modular to ease working with code. However, general functionality isn't built in. Layout is entirely static. Consider this code as the general foundation for the Receiver dashboard.

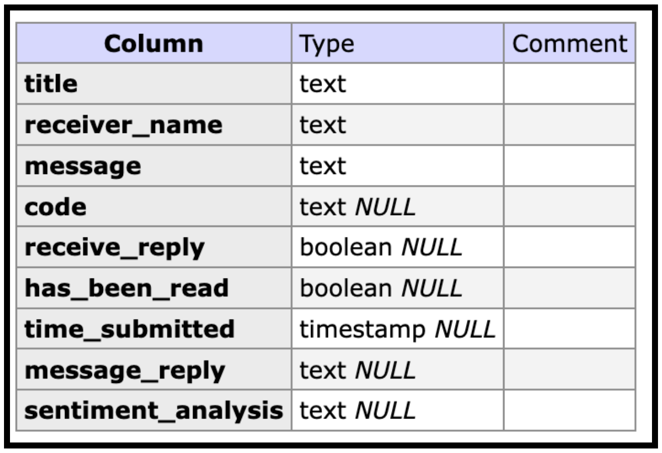
We had wished to use a sentiment analysis model and indicate the feedback visually on each message. Currently all messages have a fixed MUI positive sentiment analysis. Consider using a pretrained sentiment analysis model, such as **bert** to achieve this. This particular model is trained on 6 different languages and returns a 1-5 star rating. This could be related to 5 different MUI sentiment analysis icons.

There's a bit of empty space that isn't utilized. We thought of displaying different metrics, such as a barplot tracking sentiment of messages over specified time or a graph of feedback frequency to fill in this space. However, justifying the space with these metrics and lengthier messages might become problematic.

Database Structure

We highly recommend that the future development team contacts CAT to create a dummy database for development purposes.

The structure of the current SAFE database is as follows:



Column	Type	Comment
title	text	
receiver_name	text	
message	text	
code	text *NULL*	
receive_reply	boolean *NULL*	
has_been_read	boolean *NULL*	
time_submitted	timestamp *NULL*	
message_reply	text *NULL*	
sentiment_analysis	text *NULL*	

- **title** - Subject of the feedback
- **receiver_name** - Default to "PSU CS Department"
- **message** - Feedback body
- **code** - The unique code of each feedback message, this attribute will only be generated in features on the development branch.
- **receive_reply** - This attribute will only be generated in features on the development branch. "T" means true - Sender requested reply of their feedback. Otherwise default to "F" false.
- **has_been_read** - This attribute will only be generated in features on the development branch. "T" means true - Receiver has read the feedback. Otherwise default to "F" false.
- **time_submitted** - Time stamp of the feedback submitted.
- **message_reply** - This attribute will only be generated in features on the development branch. The Recipient's reply of individual feedback will be stored in this attribute.
- **sentiment_analysis** - This attribute will only be generated in features on the development branch. Sentiment analysis rating of the feedback message.

GitHub Branches

Main Branch

Please DO NOT modify the codebase in the main branch until the Sponsor determines that the new release version is ready for deployment to the SAFE domain (feedback.cs.pdx.edu)

This branch enables students to submit anonymous feedback to the department chair of the PSU CS department. After successfully submitting their message, the system will insert the message into the "Message" database, send an email containing the anonymous message to the desired email address (currently Mark Jones), and display a modal to the user, confirming that their message has been successfully sent to the PSU CS department chair. The page will refresh automatically whenever the user clicks elsewhere.

However, if the REST API server is not running, the message will not be able to be inserted into the database or sent to the desired email address. In such a case, a modal will pop up, notifying the user that there is an issue with the system and requesting them to try again later. The page will not refresh, ensuring that their message remains in the same place in case they wish to copy it elsewhere.

This branch includes two lines of code that differ from the MVP branch as seen below. These lines exist only in the main branch.

1. src/safeMessageDB/server.ts
    - Listening localhost address

```
// Start the server
app.listen(3001, '127.0.0.1', () => {
  console.log(`Server listening on port 3001`);
});
```

2. src/safeUI/Components/submitForm.tsx
    - Fetching SAFE domain

```
fetch(`https://feedback.cs.pdx.edu/addMessage`, {
 method: 'POST',
 headers: {'Content-Type': 'application/json'},
```

MVP

This MVP branch serves as a snapshot of the delivered project. It allows for development and testing of new changes and functionalities. It essentially offers the same features as the main branch, with the exception that the codebase can only be deployed to your personal PSU web page instead of the SAFE domain.

This branch includes two lines of code that differ from the MVP branch.

1. src/safeMessageDB/server.ts

   - Listening to rita IP address

```
const port = 3001;
app.listen(port, '131.252.208.28', () => {
  console.log(`Server listening on port ${port}`);
});
```

2. src/safeUI/Components/submitForm.tsx

   - Fetching rita server IP address

```
const port = 3001;
fetch(`http://131.252.208.28:${port}/addMessage`, {
 method: 'POST',
 headers: {'Content-Type': 'application/json'},
```

Development Branch

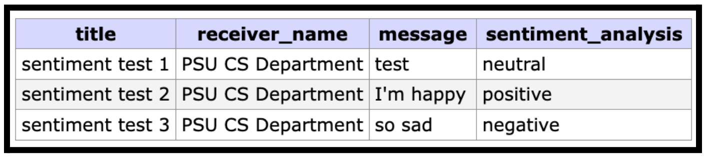
This development branch is dedicated exclusively to incremental development, as well as the staging area for prototypes to be developed in new `feature/<branch_name>` branches. These additional features are built for the `MVP` and then potentially added to the `main` branch. We highly recommend that the future development teams initiate their work from this branch, as it serves as a reference point for suggested future enhancements that the sponsor may request.

To ensure the integrity of the live website and production database, we advise the future development teams to reach out to CAT to create an additional development database to test changes against during their development process. This precautionary measure will prevent possible mishaps from occuring in the production environment.

Currently, there are a few completed features that have not yet been published to the release branch (`main`) due to the need for further testing and improvements.

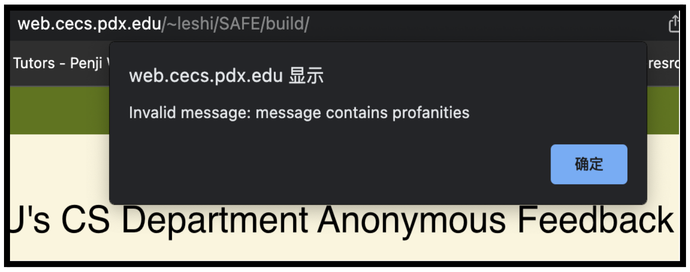
In the following sections you find a number of features that are mostly complete, but need more work to bring them out of their prototyped or alpha stage to a more solid beta candidate prior to being certified as a release candidate feature.

Sentiment Analysis



title	receiver_name	message	sentiment_analysis
sentiment test 1	PSU CS Department	test	neutral
sentiment test 2	PSU CS Department	I'm happy	positive
sentiment test 3	PSU CS Department	so sad	negative

This feature detects if the Sender is attempting to include a profanity word in the message body section and performs sentiment analysis on contents of the message. The system will provide scores based on the message to determine if it has positive, neutral, or negative sentimentality. The resulting sentiment analysis score will be inserted into the database attribute titled `sentiment_analysis`.



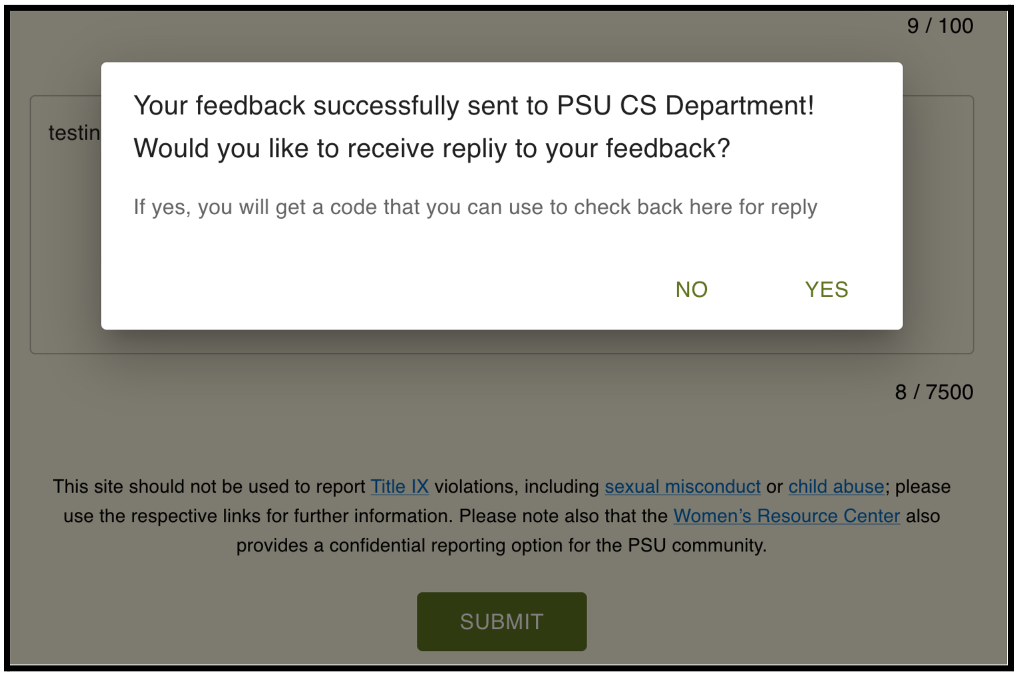
web.cecs.pdx.edu/~leshi/SAFE/build/
Tutors - Penji
web.cecs.pdx.edu 显示
确定
J's CS Department Anonymous Feedback

Invalid message: message contains profanities

In the event a profanity word is detected, the system currently prompts the Sender with an alert box. However, we recommend utilizing an alternative method, such as displaying a modal or providing helper text (such as a red colored alert message) below the message body, to inform the user that foul language will not be allowed to be submitted to the system. This will help maintain a certain degree of decorum and professionalism on the Senders part as well as a more user-friendly and intuitive experience.

<u>Subsequent action after clicking “Submit”</u>

- A modal will be displayed to inform the user that their message has been sent out successfully. The modal will also provide an option for the user to indicate whether they would like to receive a reply from the Receiver (CS Department Chair Mark Jones, for now).



9 / 100

testin

Your feedback successfully sent to PSU CS Department!
Would you like to receive repliy to your feedback?

If yes, you will get a code that you can use to check back here for reply

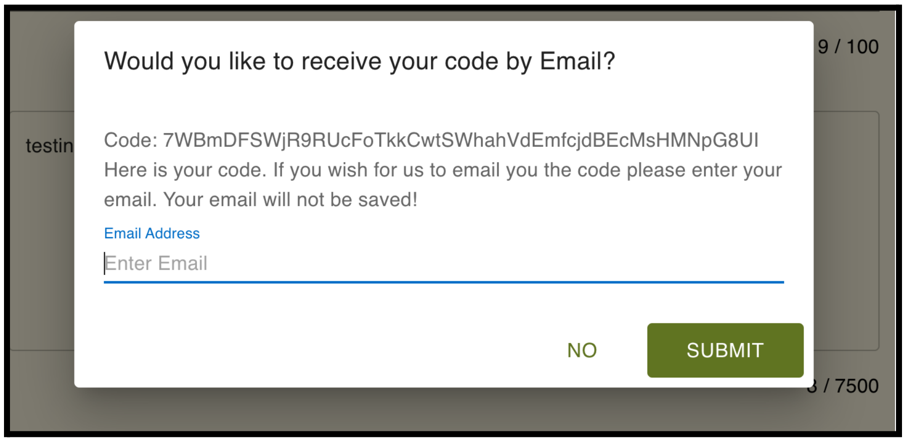
8 / 7500

This site should not be used to report Title IX violations, including sexual misconduct or child abuse; please use the respective links for further information. Please note also that the Women's Resource Center also provides a confidential reporting option for the PSU community.

- If the user selects the "NO" option on the modal, the page will refresh and no unique code will be displayed. In addition, the database attribute `receive_reply` will remain 'F' (false).

- If the user selects the "Yes" option on the modal, we will proceed to another modal where the database attribute `receive_reply` will be updated to 'T' (true).

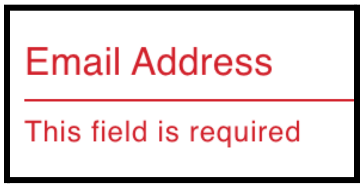
- Display a unique code when Sender requests a reply form the Receiver.



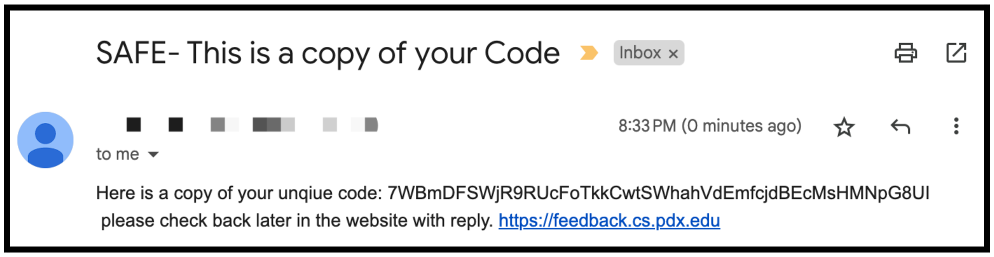
Would you like to receive your code by Email?

Code: 7WBmDFSWjR9RUcFoTkkCwtSWhahVdEmfcjdBEcMsHMNpG8UI
Here is your code. If you wish for us to email you the code please enter your email. Your email will not be saved!

- Send the provided unique code to Sender's provided email if requested.
    - If the Sender selects "NO", the unique code will not be sent and the page will be refreshed.
    - If the Sender submits the form without entering an email address, an error text alert will be displayed below the email input box to inform the user that the email address field is required.



- If the Sender submits the form with a valid email address, the system will send the unique code to the provided email address and refresh the page. Below is a an example of what this looks like currently:



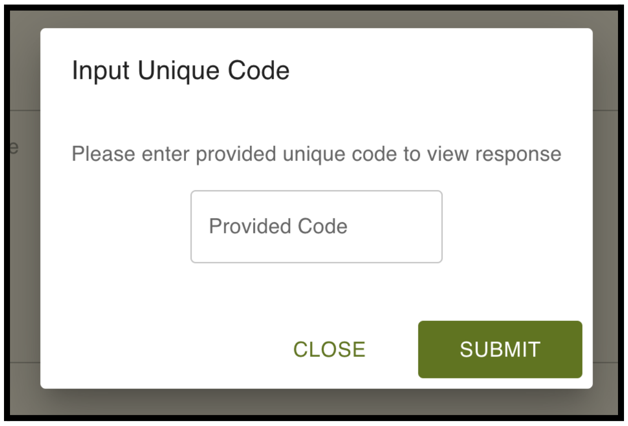
SAFE- This is a copy of your Code

[redacted]
to me
8:33 PM (0 minutes ago)

Here is a copy of your unqiue code: 7WBmDFSWjR9RUcFoTkkCwtSWhahVdEmfcjdBEcMsHMNpG8UI
please check back later in the website with reply. https://feedback.cs.pdx.edu

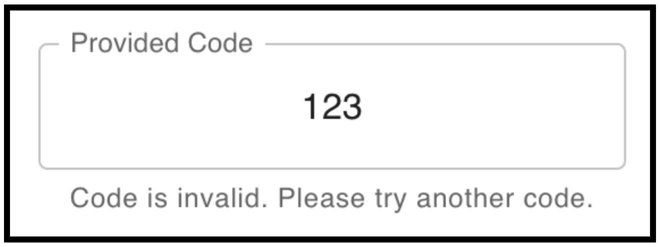
“Check Reply” button in the top right corner of Sender UI

Senders can check if a reply has been provided to their message by entering the unique code associated with their message.


Input Unique Code

Please enter provided unique code to view response

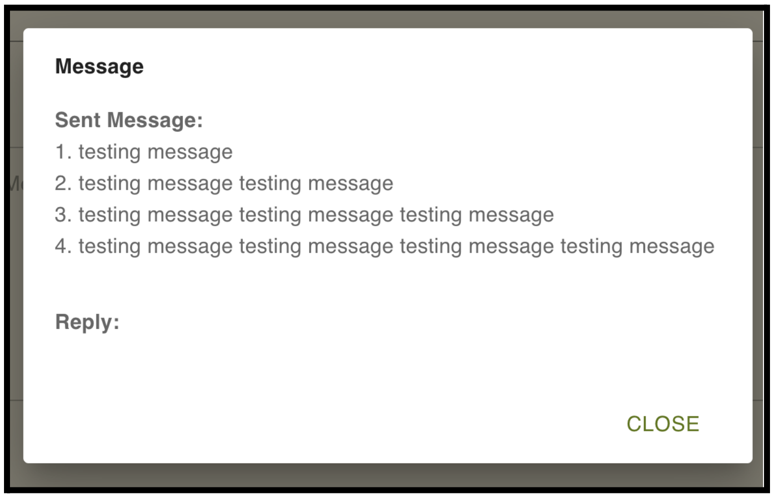
If there is no match for that unique code they provided in the database, an invalid text prompt will be displayed.



123

Code is invalid. Please try another code.

However, if there is a match for unique code in the database, a modal will be displayed with a corresponding and correct message that matches that unique code. The size of the modal will vary depending on the text.


Message

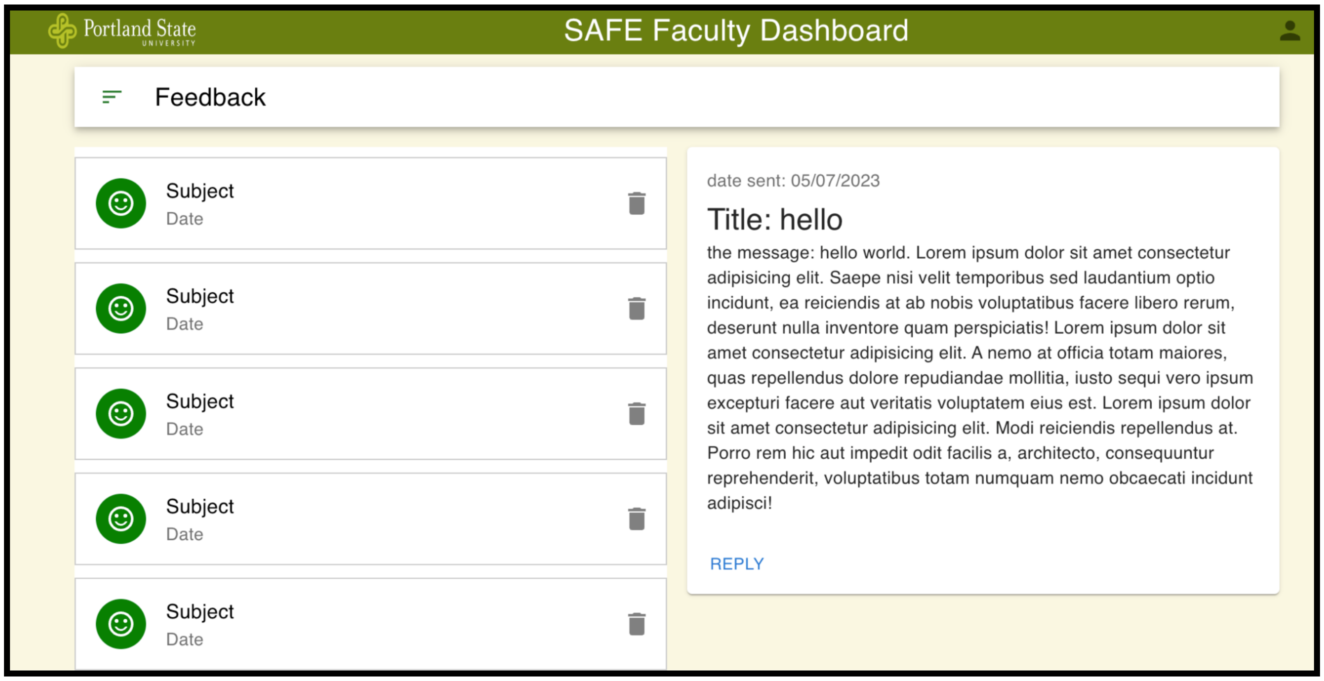
Sent Message:
1. testing message
2. testing message testing message
3. testing message testing message testing message
4. testing message testing message testing message testing message

Reply:

Receiver dashboard UI and back-end functionality

The Receiver dashboard UI has been fully designed and implemented. However, none of the buttons are currently functional due to insufficient time for integrating the UI with the back end. The back-end functionalities for deleting and replying to messages appear to be ready, but it is strongly recommended to review and validate them thoroughly.

To view the dashboard with `npm run start`, you will need modify the `index.tsx` to generate the dashboards content instead of the Sender UI.



date sent: 05/07/2023

Title: hello

the message: hello world. Lorem ipsum dolor sit amet consectetur adipisicing elit. Saepe nisi velit temporibus sed laudantium optio incidunt, ea reiciendis at ab nobis voluptatibus facere libero rerum, deserunt nulla inventore quam perspiciatis! Lorem ipsum dolor sit amet consectetur adipisicing elit. A nemo at officia totam maiores, quas repellendus dolore repudiandae mollitia, iusto sequi vero ipsum excepturi facere aut veritatis voluptatem eius est. Lorem ipsum dolor sit amet consectetur adipisicing elit. Modi reiciendis repellendus at. Porro rem hic aut impedit odit facilis a, architecto, consequuntur reprehenderit, voluptatibus totam numquam nemo obcaecati incidunt adipisci!

Useful Source References

CAT - Creating Web Pages in your Account: https://cat.pdx.edu/services/web/account-websites/

Express: https://expressjs.com/

Firefox MDN: https://developer.mozilla.org/en-US/

Git/GitHub branching standards & conventions: https://gist.github.com/digitaljhelms/4287848

Material UI: https://mui.com/

Node.js: https://nodejs.org/en

Npm: https://www.npmjs.com/

PM 2: https://pm2.keymetrics.io

reCAPTCHA V3: https://developers.google.com/recaptcha/docs/v3

Sentiment Analysis: https://huggingface.co/cardiffnlp/twitter-roberta-base-sentiment

TypeScript: https://www.typescriptlang.org/

Glossary

CAPTCHA: A type of challenge-response test used in computing to determine whether the user is human.

CAT: Computer Action Team at Portland State University. The CAT provides IT support throughout Portland State University Maseeh College of Engineering and Computer Science.

Computer Science (CS): The study of the principles and use of computers.

ESLint: Open source tool that helps find and fix problems with code in real-time.

Express: Framework used to design and build web applications quickly and efficiently.

Feedback: A message sent by Senders to the system for Receivers to eventually read and process.

JSON (JavaScript Object Notation): A lightweight data-interchangeable format that is easy for humans to write and for computers to read.

Material UI: An open-source React component library that implements Google's Material Design.

- **Google's Material Design:** A design system made by Google to assist teams in creating high-quality digital experiences for various mobile platforms and web applications.

MCECS: Maseeh College of Engineering and Computer Science.

MIT License: A permissive free software license.

Minimum Viable Product (MVP): A product with enough features to attract an early-adopter sponsor and validate a product idea early in the product development cycle.

Modal: An element on a webpage that displays on top of and disables all other content on the page.

Node.js: A platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications while providing a large number of libraries for ready-to-use situations.

Node Package Manager (npm): A package manager for the Node.js platform that allows developers to install and use various pre-built API.

PM2: A process manager that can ensure an application is always running, even when the user isn't actively logged in.

Portland State University (PSU): The location in which the SAFE application is being developed.

POST request: Method to send information from a frontend interface to a backend environment or server.

PostgreSQL: A open-source object-relational database system used to quickly store and retrieve information on systems of scale.

Prettier: An opinionated code formatter.

React: A declarative, efficient, and flexible JavaScript library for building user interfaces.

reCAPTCHA: A Google provided service to validate a user's actions on a form for a webpage.

Receiver: Person(s) who are authorized to read, reply, and manage feedback in the SAFE system.

Receiver Dashboard: The location in which the Receiver(s) will be able to engage with the system to read and/or reply to feedback and manage all feedback messages.

REST API: An interface that allows two computing systems to communicate and exchange information across a network safely and securely.

SAFE: System for Anonymous Feedback.

Sender: People who send an anonymous message to the Receiver.

Source Code: Human readable text that is compiled into a program that can be executed.

TypeScript: A strong typed programming language that builds on the already existing language JavaScript. TypeScript fully extends JavaScript and adds additional features to ensure type safety.

Uniform Resource Locator (URL): The address of a web page.

User Interface (UI): The User Interface is the point of human-computer interaction and communication in a device.