|  |  |
| --- | --- |
| APDS POE | Student Name: Karl Dicks  Student Number: 17667327  Course: BCAD3  Subject: APDS7311  Lecturer: Bruce Swart  Assignment: POE  Due Date: 31/08/2020 |

Contents

[**Introduction** 2](#_Toc46997091)

[**Security Overview** 3](#_Toc46997092)

[**Login Security** 5](#_Toc46997093)

[**Required Imports** 5](#_Toc46997094)

[**Login Method** 6](#_Toc46997095)

[**Posts Security** 8](#_Toc46997096)

[**Check-Auth Middleware** 8](#_Toc46997097)

[**Set HTTP Headers (CORS)** 9](#_Toc46997098)

[**Securing Data in Transit** 10](#_Toc46997099)

[**Require HTTPS and Self-Signed Certificates and Keys** 10](#_Toc46997100)

[**Creating the Private Key and Certificate** 12](#_Toc46997101)

[**Securing Data at Rest and Sanitizing Input** 13](#_Toc46997102)

[**Use Bcrypt to Hash and Salt Passwords** 13](#_Toc46997103)

[**Error Handling** 15](#_Toc46997104)

[**Input Validation and Registration Security** 17](#_Toc46997105)

[**Website Functionality** 19](#_Toc46997106)

[**Registration** 19](#_Toc46997107)

[**Login** 20](#_Toc46997108)

[**Google reCapture** 21](#_Toc46997109)

[**Two Factor Authentication** 22](#_Toc46997110)

[**Posts Listing** 23](#_Toc46997111)

[**Post View** 24](#_Toc46997112)

[**Download Post** 25](#_Toc46997113)

[**Edit Post** 26](#_Toc46997114)

[**New Post** 27](#_Toc46997115)

[**Departments Listing** 28](#_Toc46997116)

[**New Department** 29](#_Toc46997117)

[**Conclusion** 30](#_Toc46997118)

[**References** 32](#_Toc46997119)

# **Introduction**

APDS 7311 POE is the practical implementation of Task 1, which described the different ways in which a government posts system could be secured, and included implementation details, which have now been used when developing the system.  
APDS7311 POE has incorporated all outlined security measures, which has been submitted with this document.

This document has been created in order to describe the ways in which the system has been secured, with the use of various security methods and techniques.  
The application code will be presented, with a description of its function and how it ties in with the rest of the system for each area of the provided marking rubric.

A demonstration video will also be provided, which describes how the code can be run, and details all functions that the website can complete.

In this documentation the following aspects will be described in detail:

* The security of the login and registration process.
* The security of data at rest and in transit, including:
  + Bcrypt to hash and salt passwords for secure storage
  + HTTPS to secure transmissions of data over insecure networks
* Error handling in a safe manner – providing enough detail, but not too much as to compromise security.
* Input validation to ensure incomplete or corrupt data is not entered into the system. This also ensures whitelisting of data is completed. For example, an email address can only have certain characters in the input form. Input validation also prevents users from submitting potentially harmful scripts into the system.
* Website Functionality

The above is just an overview of the security features of the system, and the following sections contain a more detailed explanation of why and how these features were implemented in the government posts web application.

# **Security Overview**

The government website has been developed in such a way as to mitigate the following potential attacks:

* ***Brute Force Attacks***

Brute force attacks have been mitigated with the use of four primary methods, including express-brute, and express-rate-limit middleware, which both act to limit the number of times that the user can call the backend API within a certain amount of time.  
The third brute force prevention mechanism is the use of Google reCapture, which makes it difficult for bots, yet easy for humans to log in.

The fourth mechanism I have implemented to prevent brute force attacks is Firebase Authentication One Time Pin’s. This only allows users who have signed up and have their phone with them to log in, as it sends their phone a one-time pin, which cannot be re-used.

* ***Session Jacking***The web application uses a combination of secure cookies and JWT’s to store session information. The system stores the JWT in the secure cookie, and therefore limits access to the JWT, as the cookies are set to “HttpOnly” and “Secure”, and also have an absolute timeout of one hour. The JWT also has the same absolute timeout.  
  In addition to these methods and techniques employed, HTTPS is enforced, with a self-signed certificate (generated by OpenSSL), which improves security of data in transit, with the use of encryption.
* ***Cross Site Scripting (XSS) and CSS Protection***

There are many ways in which XSS attacks have been prevented, including input form validation (regex), output HTML sanitization, enforcing HTTPS, and using cookies with the “HttpOnly” flag set to true.  
By validating all input in forms, the system is kept secure from potentially dangerous JavaScript injection. It does this by limiting what the user can input in each form field.

HTML sanitization is used for all output fields, in order to prevent scripts from being injected into the site, and being run by its users. For example, the web application uses “ng-bind”, which automatically sanitizes output before it is displayed on the site.

The site uses HTTPS only, so any requests to an HTTP variant of the site will not be allowed, and will be re-routed to the HTTPS site, or will not resolve. This has been accomplished with the use of a self-signed certificate.

In addition to all the above measures, XSS has been mitigated with the use of secure cookies, which are inaccessible from client-side JavaScript injection. Cookie parser middleware has been used at the backend to read the JWT’s from the HttpOnly cookies, however this is at the backend, not the front-end, which secures the site against XSS attacks.

* ***Username Harvesting***

Username harvesting has been prevented by displaying generic error messages when a user attempts to log in or register with a valid username (email), however their password is incorrect. For example, when a user attempts to log in with an incorrect password, they are not presented with a “Password is incorrect.” dialog, but rather an “Authentication Failure” message, which does not provide any valuable information to the potential hacker.  
In addition to this, each of these failed login attempts will add up to a point where the user cannot log in, and the brute force middleware prevention mechanism will kick in, to prevent users from attempting too many logins.

All of the above processes are logged using the Morgan logger, and these logs can therefore be used by system administrators to block certain IP ranges, or even single IPs if they detect unusual activity.

* ***Credential Security***  
  Credential security is ensured by randomly salting passwords, and then hashing them, so no two passwords will have the same database value. Therefore, even if a hacker gains access to the web application backend database, they will not be able to retrieve plain text password values for stored accounts.

This is accomplished by using Bcrypt, which automatically randomly salts and then hashes passwords input by the website’s users.

* ***Input Sanitation***The application uses four main methods of preventing potentially malicious code from entering the system.  
  The first method uses regular expressions to whitelist only certain characters from being input in all form fields.  
  The second method uses sanitize-html in the backend to prevent malicious code from persisting in the backend server (MongoDB).  
  The third method uses ng-bind directive and interpolation, which automatically sanitizes values before they are output.  
  The fourth and last method uses the built in Angular defence method against JavaScript injection, to prevent malicious scripts from running (DOM sanitizer).

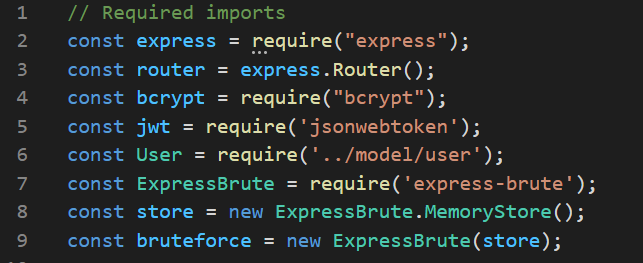
The following sections detail all the above listed security features, techniques, methods, and best-practices in my own application.

# **Login Security**

The following methods have been implemented in order to prevent malicious users from signing up, or logging in. The below features prevent Cross Site Request Forgery, Click Jacking, Brute Force attacks, Cross Site Scripting, and other such hacking techniques.

## **Required Imports**

The below image shows all required imports for bcrypt, jwt tokens, and express brute, which are all used to secure the login process.

*apds-gov\backend\routes\users.js*  


Line 4 – Import Bcrypt package into the project.  
Line 5 – Import JWT Tokens package into the project.  
Line 7 – Import Express Brute package into the project.

A description of how these packages are used to secure the system is described in the following sections.

## **Login Method**

*apds-gov\backend\routes\users.js*  


The description of this code is provided on the following page.

The login method, which is used to authenticate users when they navigate to the login page, enter their details, and click the “sign in” button, is used to generate JSON Web Tokens (JWTs), which stores user information for their “sessions” while actively using the web application. This information is stored in a cookie, with HttpOnly flag set to true, and therefore prevents JavaScript from accessing cookie information. This prevents cross site scripting (XSS) attacks, as malicious JavaScript code cannot easily retrieve the JWT session information.

The Json Web Tokens and Cookies will expire after one hour – this is an absolute timeout, so the user can log in, work for one hour on the system, and then will need to sign in again after the absolute timeout has been met (after one hour).

In addition to this, “bruteforce.prevent” is used to prevent users from unsuccessfully logging in multiple times in quick succession. It limits the number of backend requests so that malicious users, who may use a brute force attack, will be locked out and their requests will be discarded after a short period of time.

In addition to this brute force attack mitigation strategy, I have also used Express Rate Limit middleware to limit the number of API requests that are made in a certain timeframe.

When the user logs in, they will provide their password, which will need to be compared to the value stored in the database (which is both hashed and randomly salted). Bcrypt implements both of these techniques to compare the input password to the value stored in the database securely. These values are securely transmitted over HTTPS, and cannot be sent over HTTP, which further secures user data (more on this in further sections).

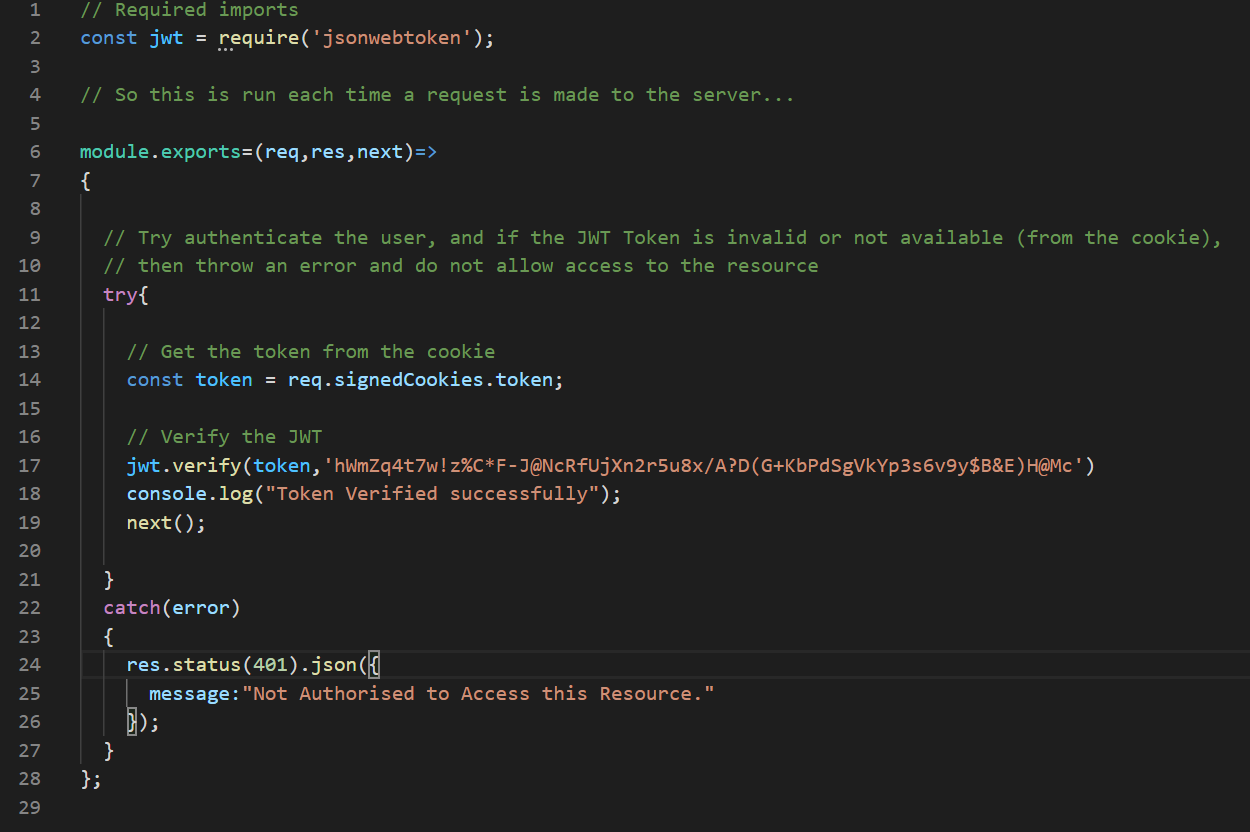
Finally, once the user has logged in, their Json Web Token will be generated and stored in an HttpOnly Cookie if they have passed all these security checks. If not, a generic “Authentication Failure” message will be displayed, which will not provide any information that could be used by potential hackers to gain access into the system. For example, it does not say “Password Incorrect”, as that would mean that the username is valid, and the hacker can use this information to gain access more easily into the system.

In addition to the above functionality, Firebase Authentication has been implemented, so that the user must complete Google reCapture, and submit an SMS OTP each time they log in. When signing up on the website, users must provide their cell phone number, which receives a One Time Pin each time they attempt to log in. If the code entered is incorrect, then they will not be granted access to the system.

Line 61 – Prevents Brute Force attacks by using the bruteforce.prevent method.  
Line 76 – Compares password input with Bcrypt hashed and salted password.  
Line 90-94 – Generates (signs) a JWT when the user logs in.  
Line 106 – Generates Cookie with JWT content (set to HttpOnly).

# **Posts Security**

## **Check-Auth Middleware**

*apds-gov\backend\middleware\check-auth.js*  


The check-auth middleware is used to handle requests for access to the API, which includes all CRUD operations – Creating, Reading / Reporting, Updating, and Deleting operations. For example, the user must be logged in and authenticated in order to view or create new posts, among other functions. The check-auth middleware retrieves the Cookie, extracts the JWT, and verifies it. If it is invalid, then then it will throw an error and direct the user to the login page, else if it is valid, then the user can complete the action.

Line 14 – Retrieves the secureCookies.token value (the JWT) from the request headers.  
Line 17 – This line of code verifies that a token is available, and that it is the same as the token which was created when the user logged in.

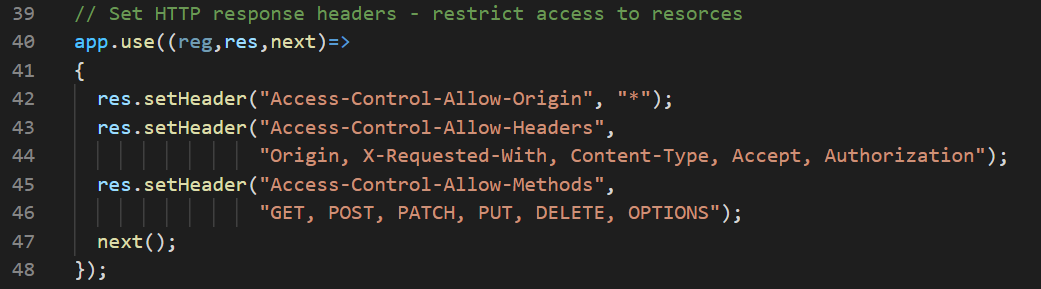
*apds-gov\backend\routes\posts.js*  


This check-auth middleware is run every time a user performs CRUD actions on the posts, including the above method used to delete posts. This ensures that the Cookie and JWT are used every time a user performs actions on the database, in turn securing the system.

Line 76 – CheckAuth run when the user attempts to delete a post.

## **Set HTTP Headers (CORS)**

*apds-gov\backend\routes\app.js*



CORS is a mechanism that uses traditional HTTP headers to allow browsers to give a website running at one origin access to selected resources from a different origin. A website executes cross-origin HTTP requests when it requests a resource from a different origin (domain, protocol, or port) from its own. (Mozilla, 2020)

CORS is blocked by default on most web servers as this provides a higher level of security, however we need users to be able to access web resources from their browsers from different origins than that of the web server.  
We therefore allow resources to be requested from other origins, but limit the actions that they can perform, by setting header and method access using the above code. (Mozilla, 2020)

CORS middleware is also used to ensure only certain actions can be processed.

By setting these headers, we are also preventing Cross Site Forgery Attacks, as we are limiting which origins can request or send data to or from the server.

# **Securing Data in Transit**

The application has been configured to only allow requests over HTTPS, which encrypts data in transit (Transport Layer Security). Data in transit is therefore encrypted before it is sent to the server, and vice versa. It uses a self-signed certificate, which was created using OpenSSL.

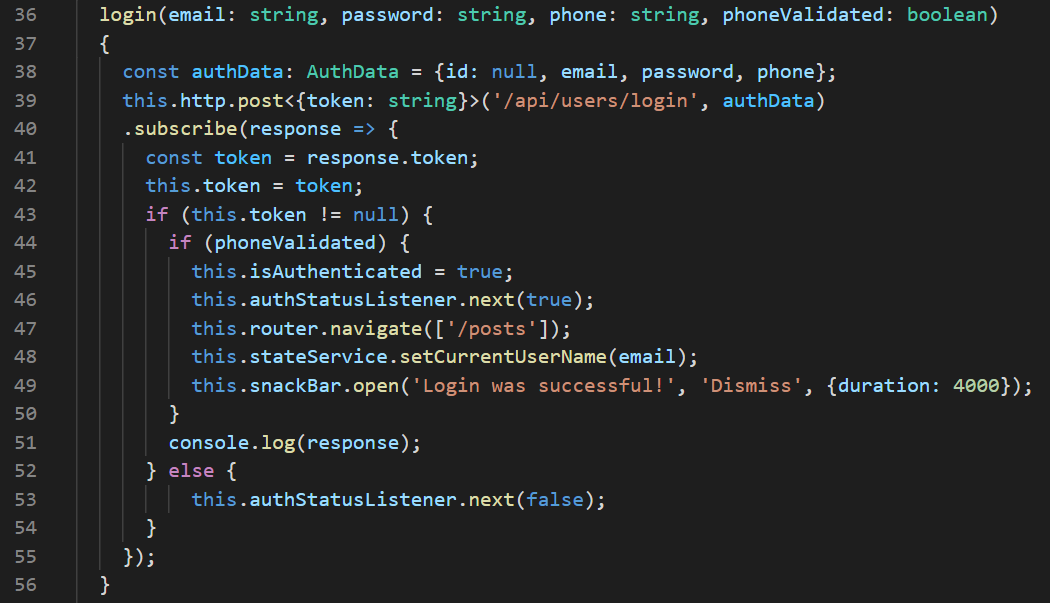
## **Require HTTPS and Self-Signed Certificates and Keys**

*apds-gov\server.js*  


The above image shows that the server is only listening and responding to requests on HTTPS, on port 3000, using a self-signed certificate.  
The client browser can therefore not request or alter information using the insecure HTTP protocol. This forces information to be transmitted over secure protocols.

Line 2 – The server uses the import of “HTTPS”, and not “HTTP”.  
Line 7-11 – Server created using the OpenSSL key and certificate, which are used to authenticate and decrypt server requests from users.

All server requests use the HTTPS protocol – an example is shown below.

*apds-gov\src\app\auth\auth-service.ts*  


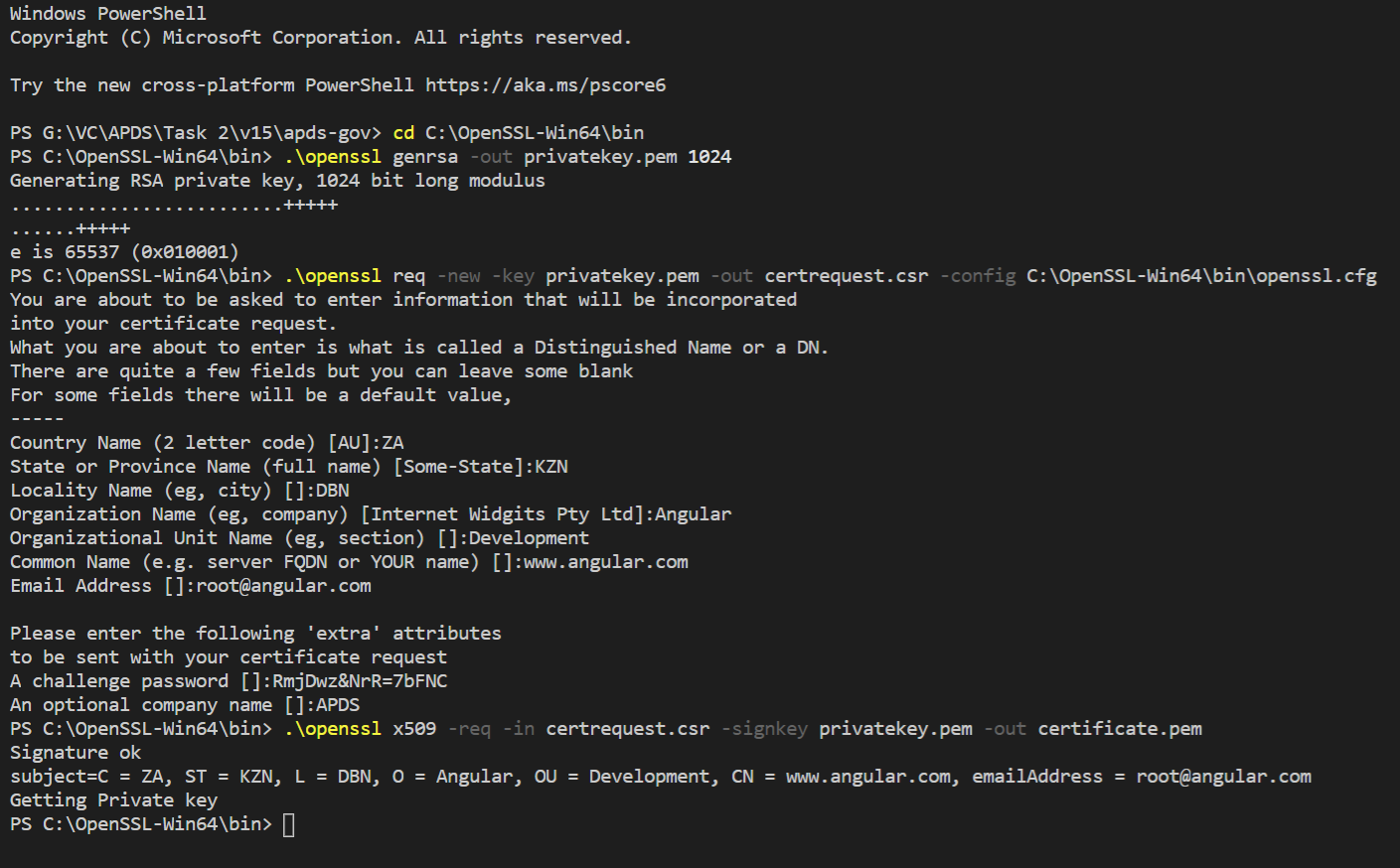
This image shows an HTTPS request for information on the apds-gov backend server.  
These requests use asymmetric encryption, digital certificates, and signatures to secure the data that is transferred from server to client and vice versa.

Line 39 – This line shows that the front end is contacting the backend server using the \*HTTPS protocol, which secures data in transit.

\*Note that the services request information from the API using a proxy, and can therefore call ”/api/users/login” instead of “<https://localhost:3000/api/users/login>”. More on this later.

The following page shows the private key and certificate being created.

## **Creating the Private Key and Certificate**

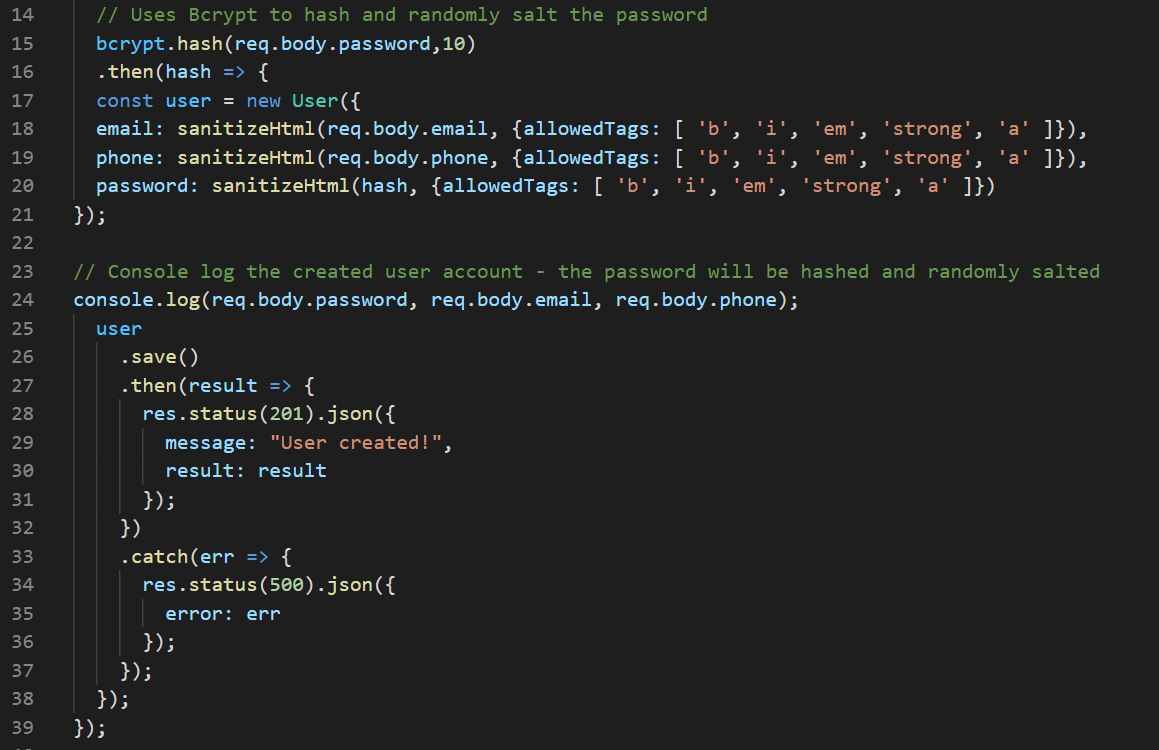


The above screenshot shows the private key being created, signed, and a certificate created. This is used to encrypt and decrypt the information sent over less secure networks, using the HTTPS protocol.

# **Securing Data at Rest and Sanitizing Input**

The backed server uses Bcrypt to both hash and randomly salt the password input when a user signs up a new account.  
When the user logs in, the password is encrypted and compared with the stored password, and therefore even if the user document is found by hackers, its contents will be of little to no value, as the original password cannot be determined from its database value.

## **Use Bcrypt to Hash and Salt Passwords**

*apds-gov\backend\routes\users.js*  


The above method is used to create new users on the system. The first process it completes when it retrieves the account password is to hash and randomly salt it using the Bcrypt method.

Line 15 – Use Bcrypt to hash and randomly salt the password which has been input by the user.

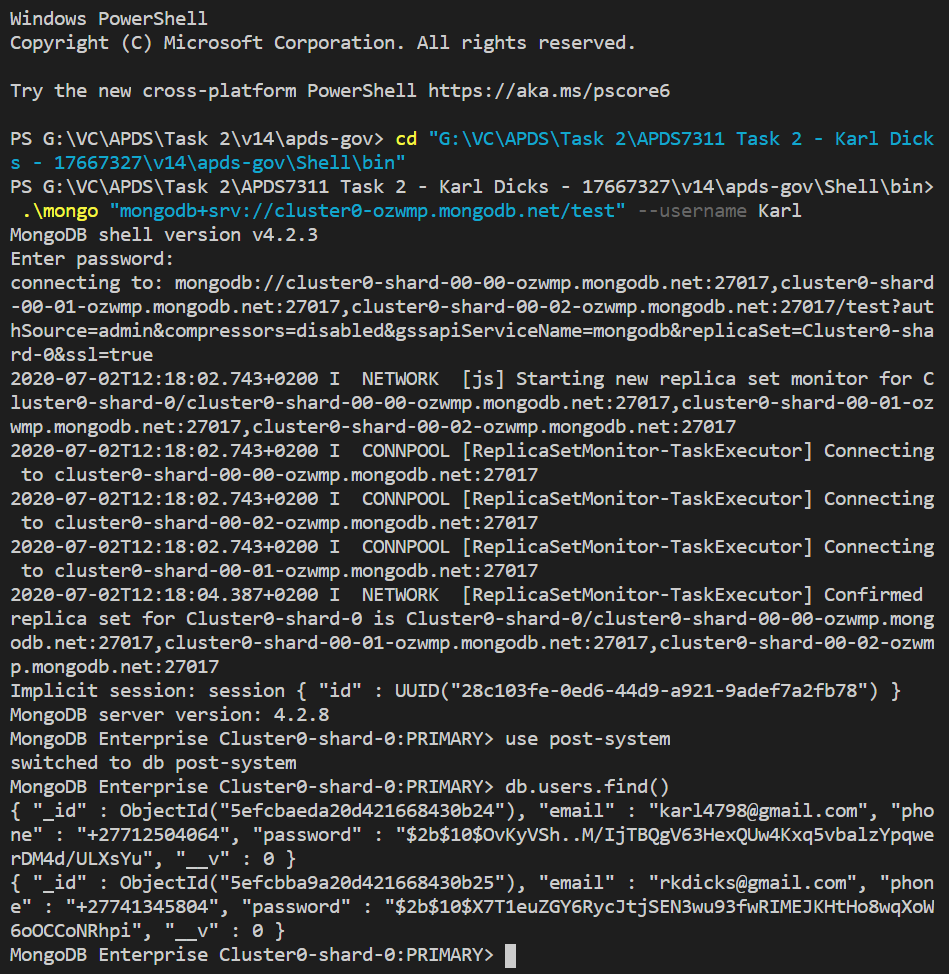
Line 18-20 – Use sanitizeHtml to only allow certain characters to persist in the database. This method is used on all inputs for the system, including posts, departments, and user creation.

In order to view these user accounts in the database, the administrator can use Mongo Shell. This process is shown on the following page. (MongoDB, n.d.)

The MongoDB Shell has been included in the root project files directory in a folder named “Shell”.

In order to view the contents of the database, the following commands can be used:

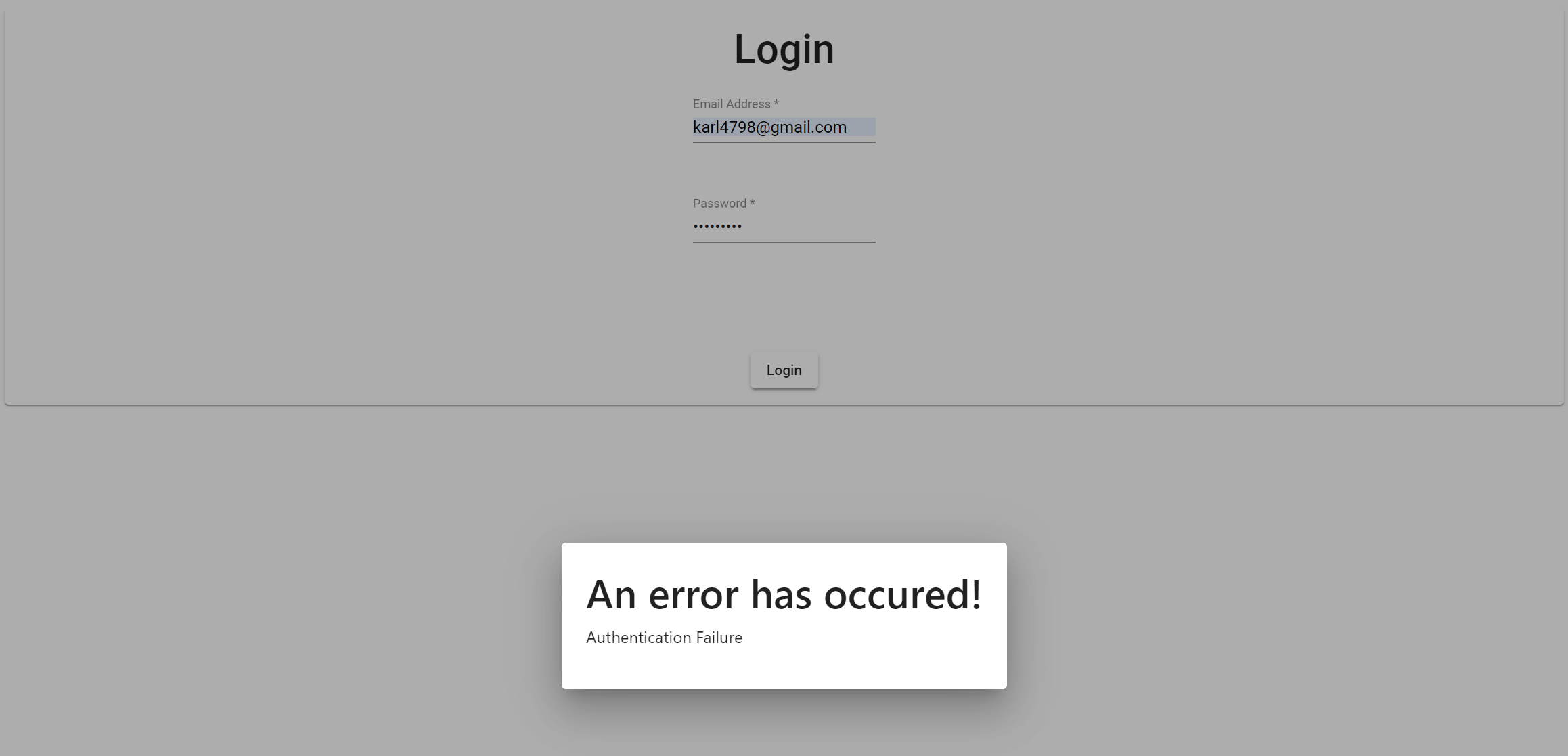
* Navigate to the Shell directory: “APDS7311 POE - Karl Dicks - 17667327\v18\apds-gov\Shell\bin”
* Connect to the database using MongoDB shell: “.\mongo "mongodb+srv://cluster0-ozwmp.mongodb.net/test" --username Karl”
* Enter the password for the database: VHl0MLhuWt2kL899
* Use “post-system”
* To view collections run the “show collections” command (w3resource, 2020)
* To view the all user accounts run “db.users.find()”.
* All users will now be shown, with their hashed and randomly salted passwords – notice that the 2 passwords are exactly the same, yet have different database values.



# **Error Handling**

All forms on the web application are validated for null, incorrect, or invalid information. An appropriate error message will be displayed when or if an error occurs, which shows as little information as possible so as to not give potential hackers any valuable information about the system or user account.  
For example, it will not show “Invalid Password”, as this will likely mean that a valid username was entered. The hacker can then try to find the matching password for the account.

As shown below, a generic “Authentication Failed” error message is displayed, which is a “safe” and suitable message displayed when a user enters invalid authentication information.



Example – Error displayed when trying to log in with incorrect login details.

*apds-gov\backend\middleware\check-auth.js*  

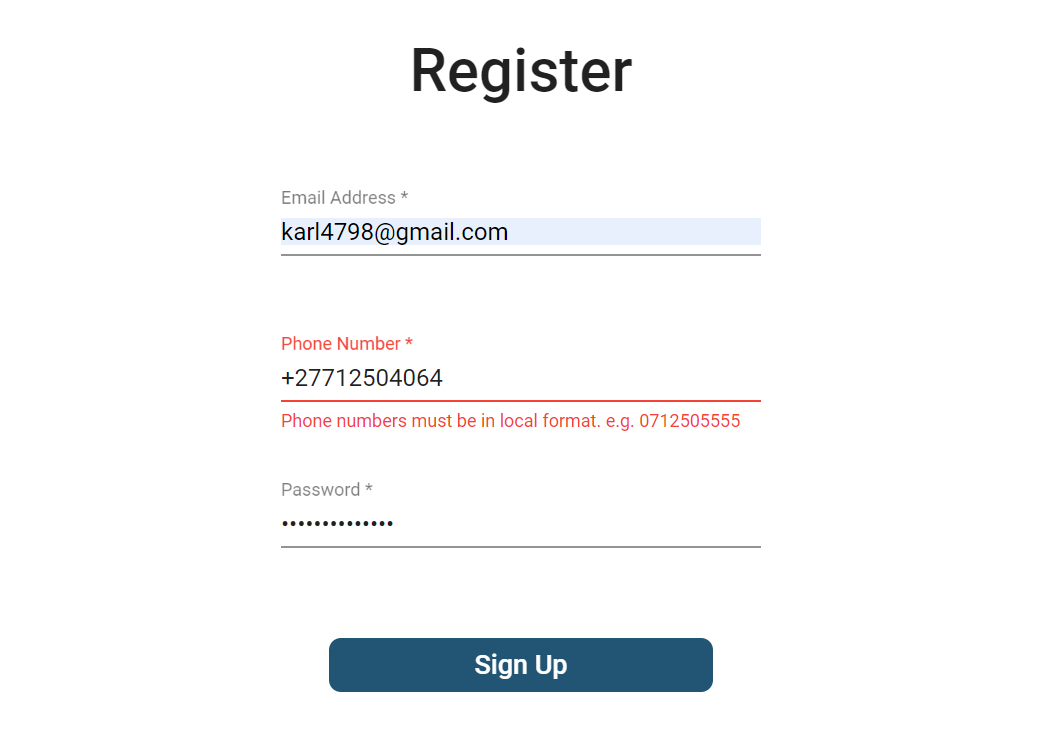

An example of a try / catch block of code is provided above, where if an error is caught, it will display the message “Not Authorised to Access this Resource”. This occurs when the user attempts to access the posts when they are not logged in and no JWT Token is found (from the cookie).

The following code shows an error interceptor, which intercepts errors and handles them appropriately, providing suitable error messages, which are shown in the previous image.

*apds-gov\src\app\error.interceptor.ts*  


Line 13 – Accepts any type of HTTP request, and if it fails, then display a suitable message, and navigate the user to the login page if they attempt to access resources without permissions (without their Cookie).  
Line 26-30 – Display a snack bar message and navigate the user to the login page if they are not authorised to access the resource.  
Line 36 – Open a dialog, which shows the error message.

# **Input Validation and Registration Security**



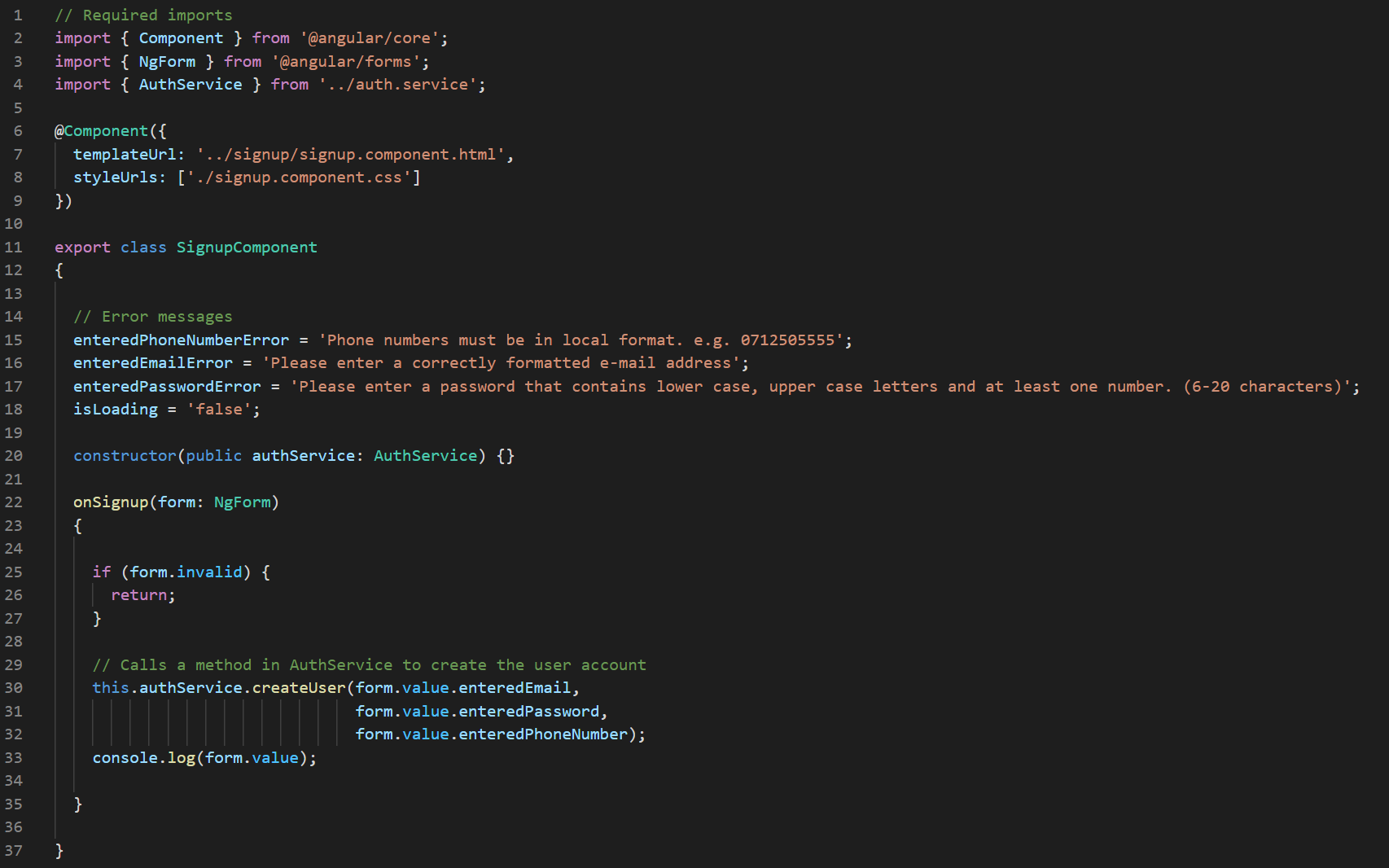
Input validation is shown on the above image, where the user is prompted to enter their Email Address, Phone Number, and Password. The password is not shown on the page, for security reasons. The password input box uses the type = “password" attribute, which allows for input, but does not display the characters when typed.

The Phone Number must be input in local format, so therefore the user cannot enter the country code (+27). The maximum number of digits is also limited to 10 digits – a normal South African cell phone number.  
The form validates input, and will not allow you to continue until all form fields are validated.

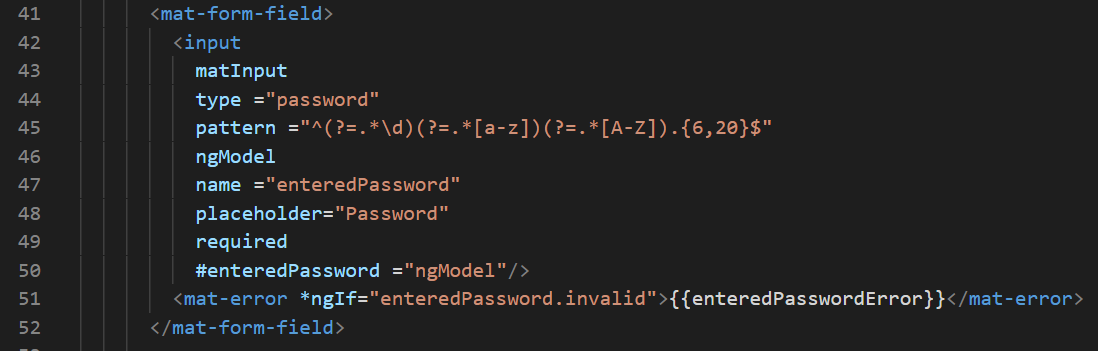
Suitable messages will be provided below form fields, and information as to how you can correct the errors will be included, as to assist the user in completing the form submission. Form validation is provided on all form fields on the application, so that the system does not end up with invalid or incomplete information persisting in the database.

The code required to implement input validation is described on the following page.

Input validation has been implemented on all forms on the web application, and one such example is provided below. The registration form uses Regular Expressions, minimum length, null, and other checks to see if input is valid. If the input is invalid, then a suitable error will be displayed, which is declared and assigned in the component TypeScript file.

*apds-gov\src\app\auth\signup\signup.component.ts*  


In the above code, error messages are declared and assigned suitable messages.

*apds-gov\src\app\auth\signup\signup.component.html*  


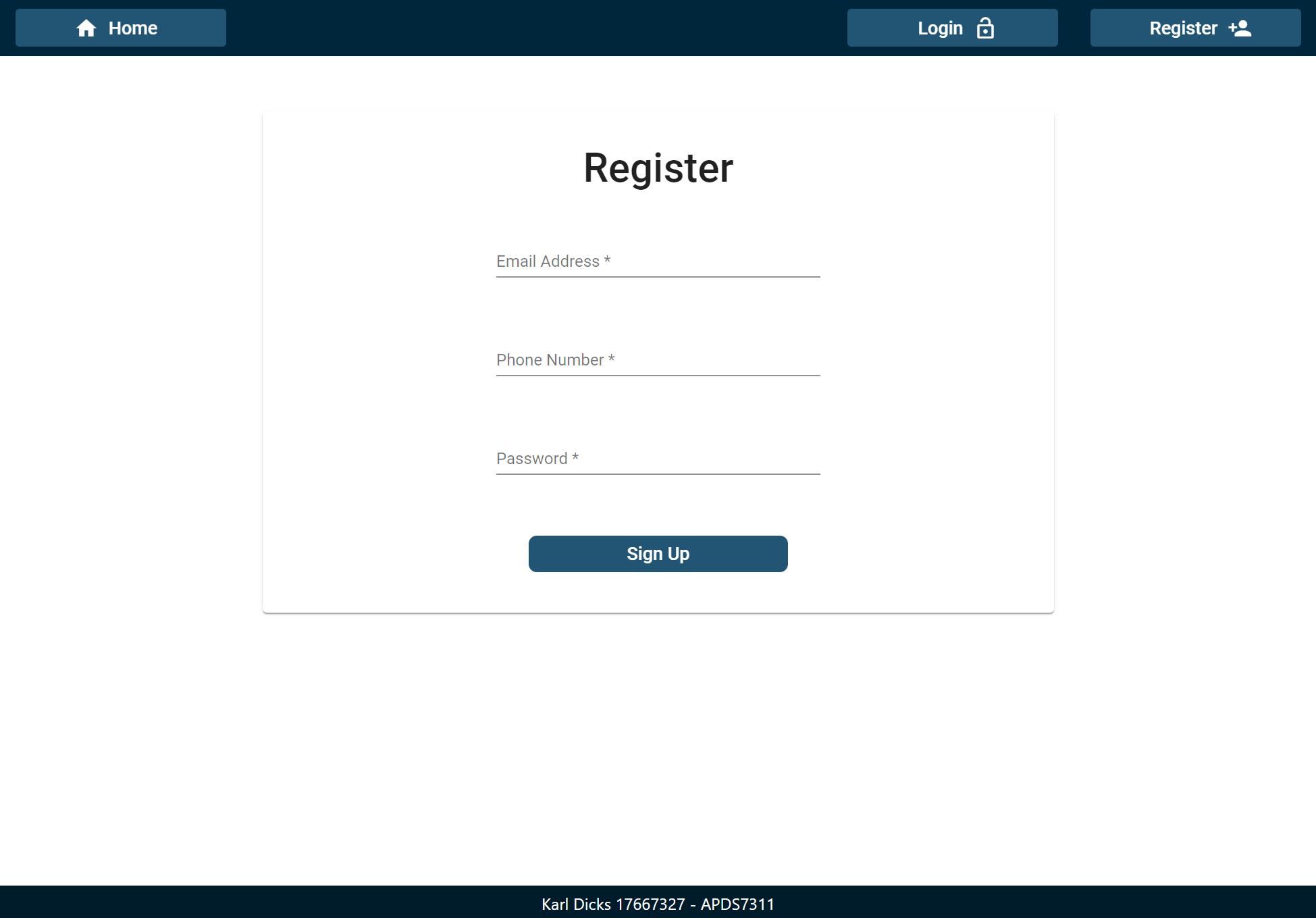
The above example is one input field, with regular expression (pattern) for password inputs, which must contain lower case letters, upper case letters, digits, and must also have a length of 6 to 20 characters. The tag “required” is used so that input cannot be null. The tag “type” is set to “password” so that input is captured but not displayed when input on the form. If the entered password is invalid (does not conform to the regular expression or any other checks), then the enteredPasswordError is displayed.

# **Website Functionality**

A brief overview of the system functionality can be found in the following section. It details what the web application can do, in order to streamline the posting of government notices.

## **Registration**

The APDS web application requires that the user register and log in to the website before viewing, editing, deleting, or downloading any posts on the website.  
Below is the registration process that the user must complete in order to access this information.

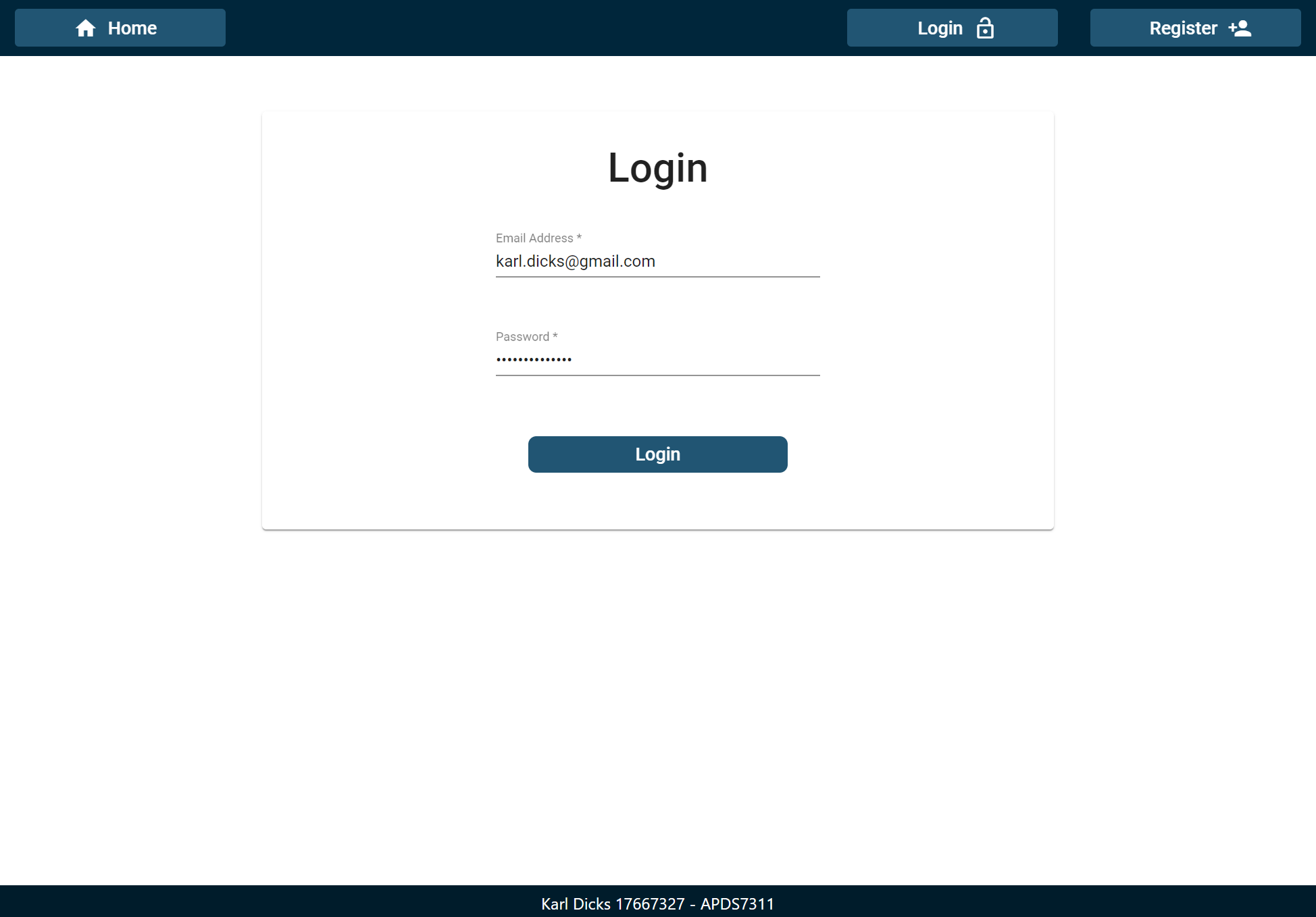


When the website loads, the user can navigate to the “Register” page and enter their details.  
Once filled in, the form can be posted by pressing the “Sign Up” button, which validates input using Regex, and other methods, and then signs up an account for the user.

Once this has been completed, the user can log in by pressing the “Login” navigation button.

## **Login**

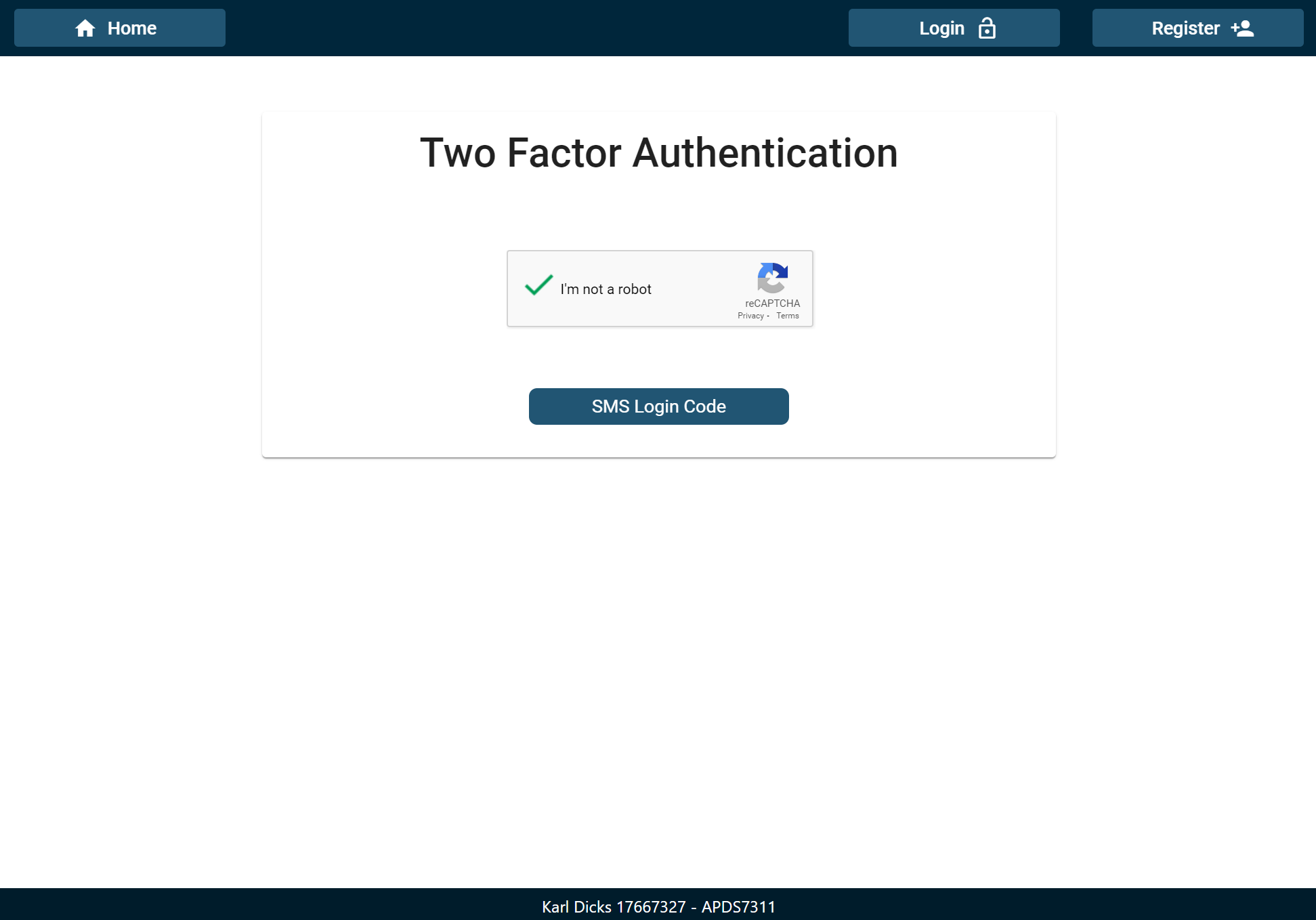
Once the user has an account with the web application, they can log in by navigating to the “Login” Page.  
Below is the login process that must take place before the user can access any information.



The user can enter their details on the above page and press the “Login” button to start the login process.

## **Google reCapture**

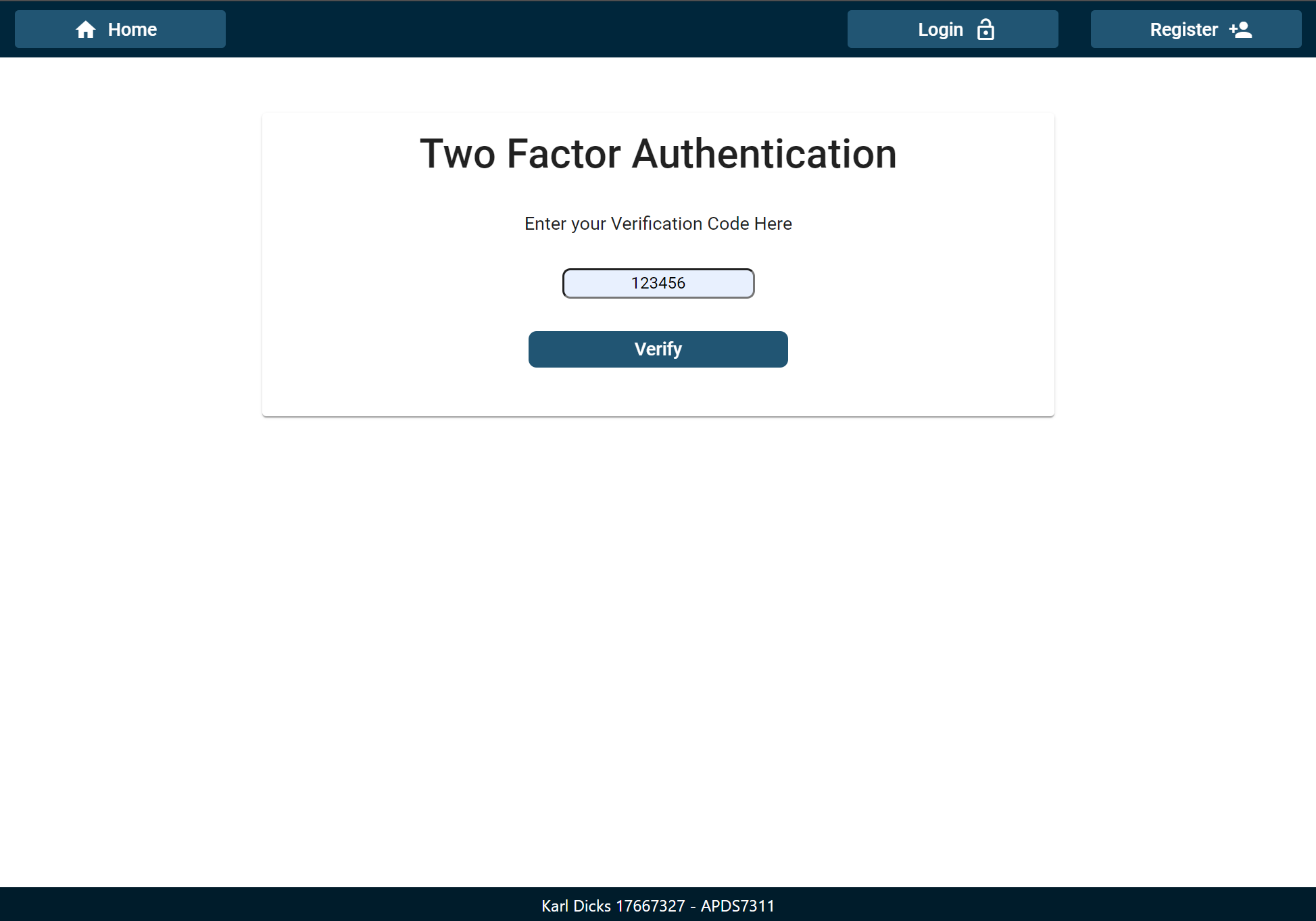
Once the user has entered their unique email address and password, and clicked the “Login” button, they will be brought to the Google reCaptcha page.



The user must now complete the reCaptcha process, which makes it difficult for bots to log in, but easy for users to log in.  
Once complete, they can click the “SMS Login Code” to receive a One Time Pin (OTP) on their phone (from Firebase Auth).  
JQuery is used on this page to replace the Google reCaptcha div element (after successful completion), with the verification entry input and button.

## **Two Factor Authentication**

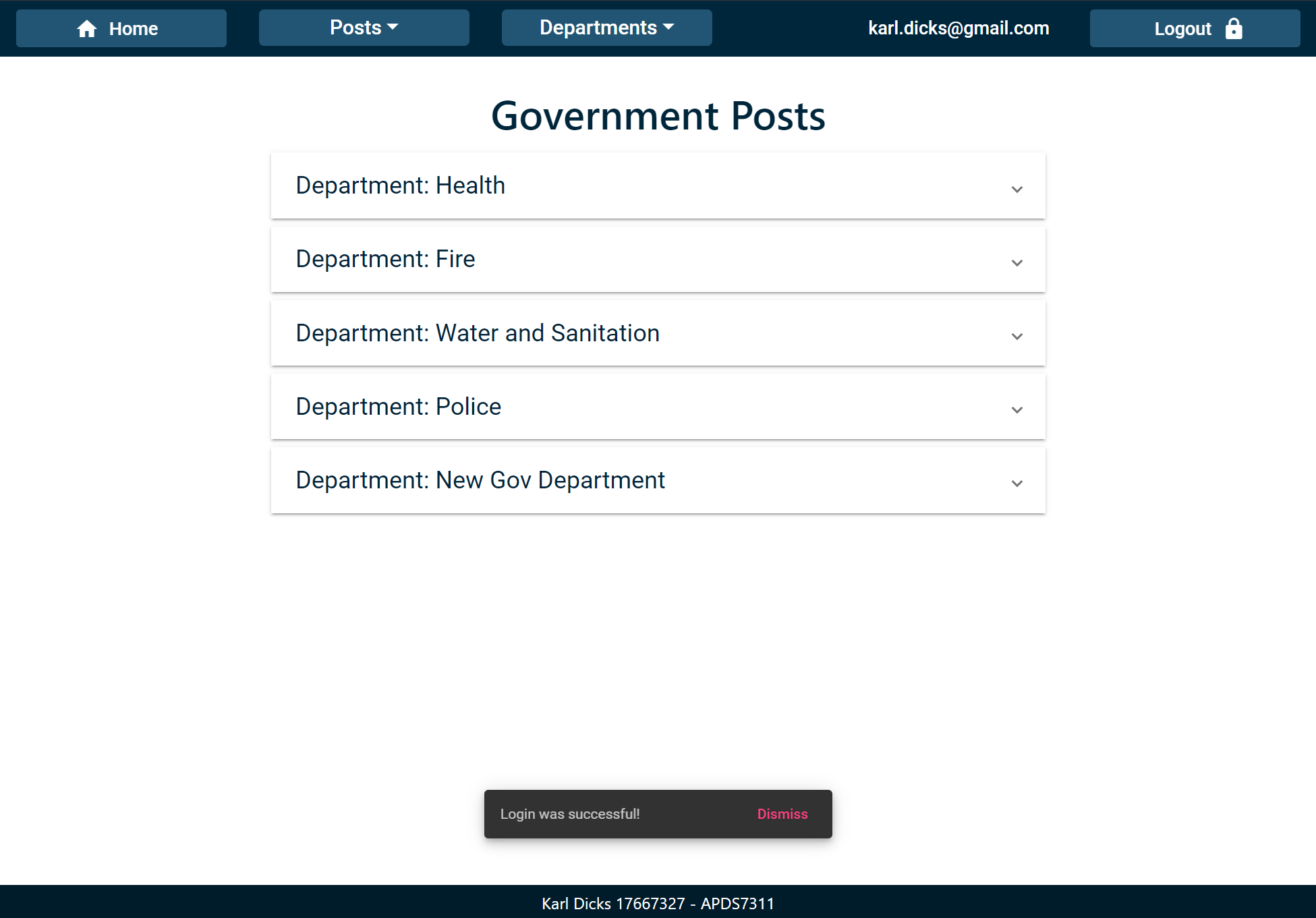
Once the reCapture has been completed, and the “SMS Login Code” button has been clicked, the user will be navigated to the Two Factor Authentication page, where they can enter the code sent to their phone.



Once the user enters the OTP code, and clicks the “Verify” button, the user will be authenticated, and then navigated to the posts page.

## **Posts Listing**

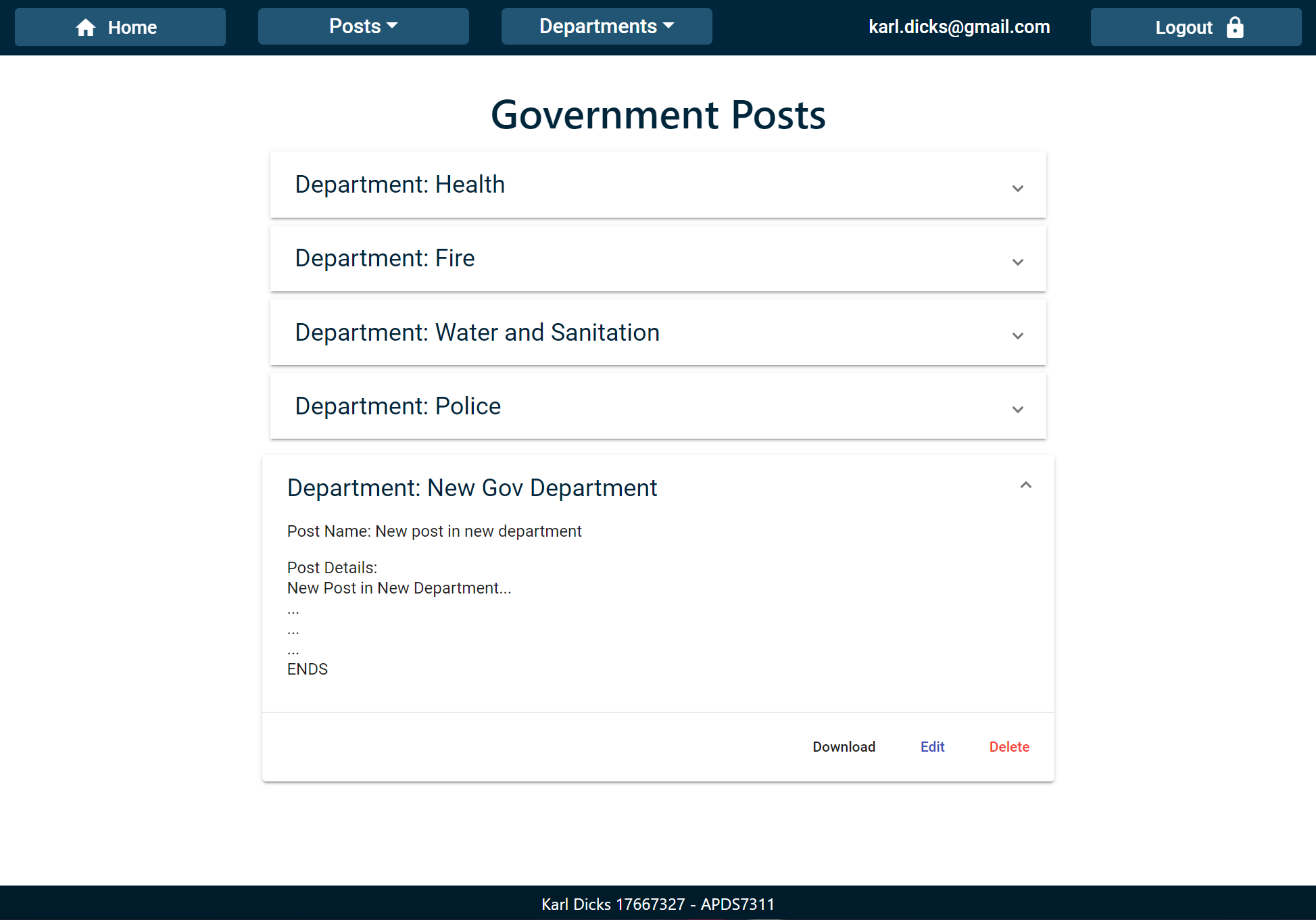
When the user logs in, they will be navigated to the post listing page, which shows all posts placed by ministers. A snackbar message will be displayed for four seconds, or the user can use the “Dismiss” button to dismiss the dialog.



Posts are displayed by department, and if clicked on, their contents will be displayed (as shown on the following page).

## **Post View**

Once the user has logged in, they may view posts, and expand their contents by pressing on the list items. These posts have download, edit, and delete functionality, which is shown on the following pages.



Once the user clicks on a post, it will be expanded with post details, including the post name, department, and details.

The user can activate different routes, including:

* Download – Downloads the post in PDF format.
* Edit – Allows the user to edit the post, and save changes.
* Delete – Allows the user to delete a post from the database.

These functions are shown on the following page.

## **Download Post**

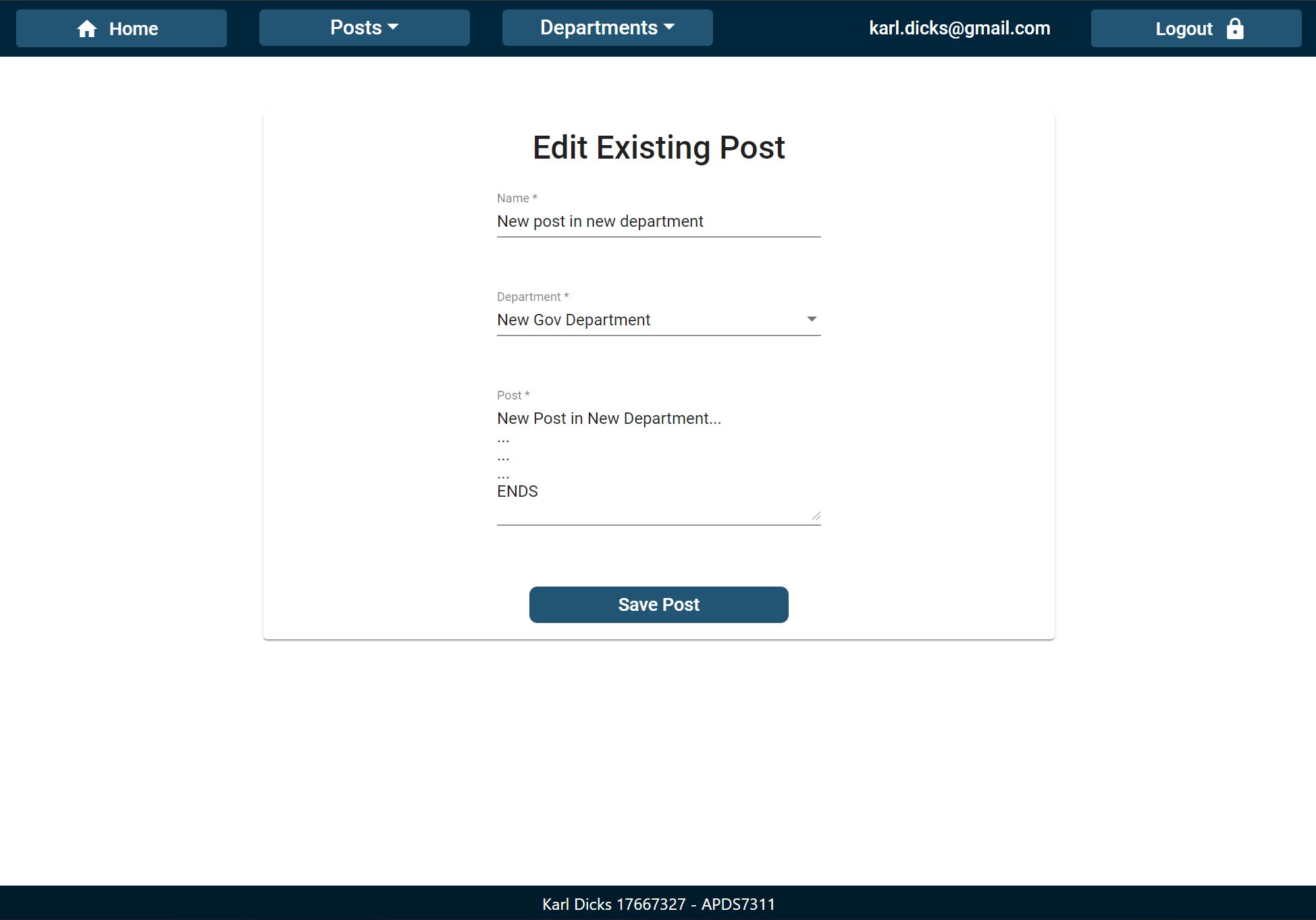
Posts can be downloaded by pressing the “Download” button on the individual post, which will download the post in PDF format. This PDF will include the post id, name, department, and content.



An example of a downloaded post can be seen above, which is in PDF format.

## **Edit Post**

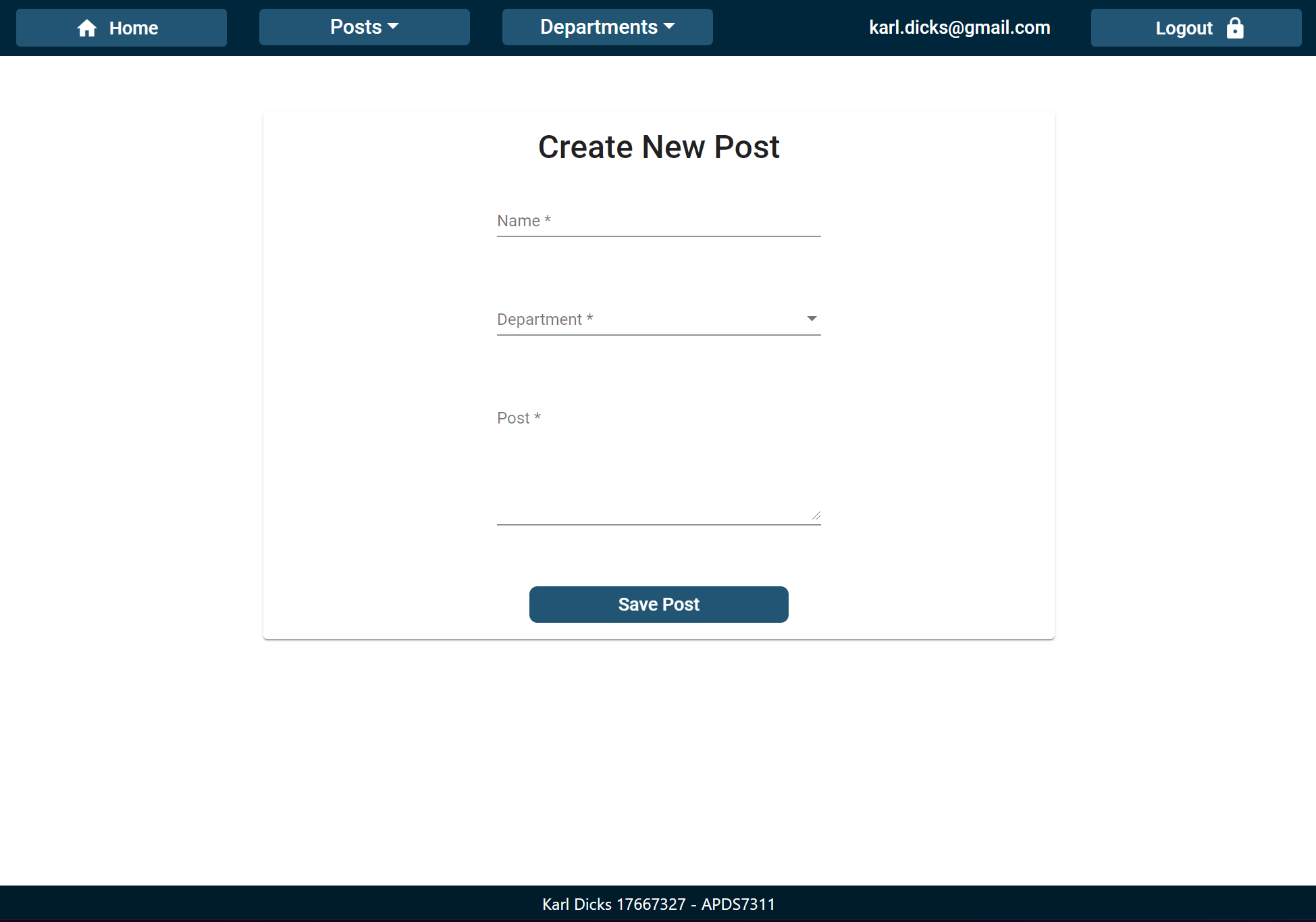
Posts can be edited by pressing the “edit” button on the individual post, and the user may edit the post name, place the post in another department (dropdown list), or edit its content.



Once the user edits the post, they may click the “Save Post” button, which will update the post in the database, and navigate the user back to the posts listing page. The changes will now reflect on the front end.

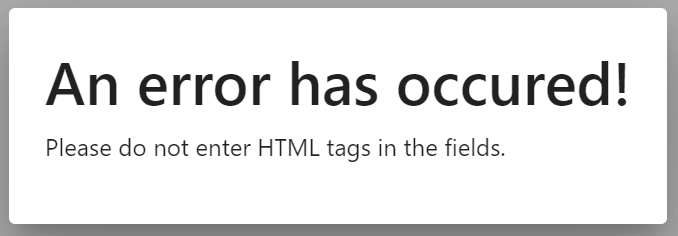
## **New Post**

The user may create a new post by navigating to the “Posts” -> “New Post” tab on the navigation bar. This will present the user with form fields which can be used to create a new government post.



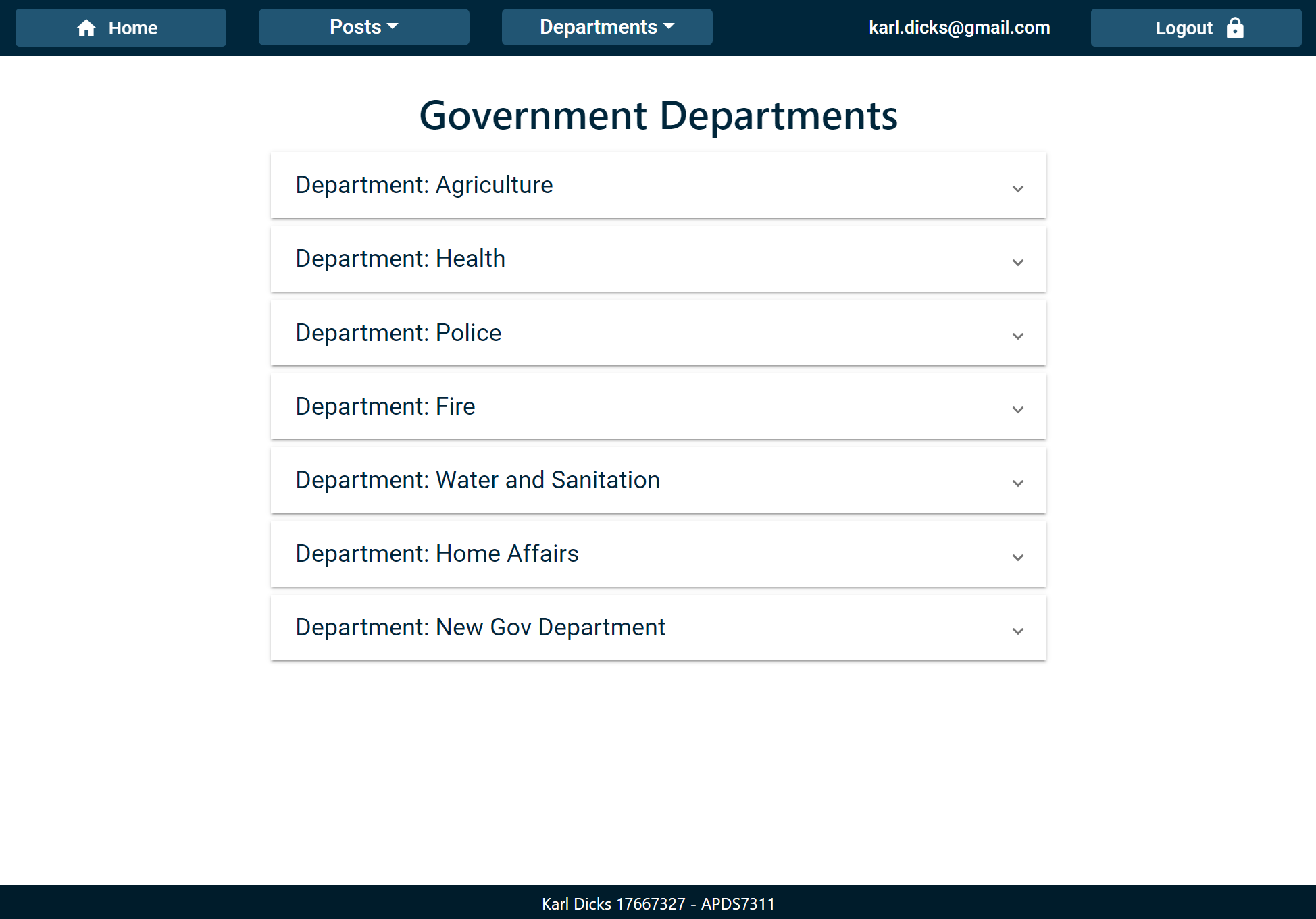
Once the user fills in all the required information, including a post name, department, and its content, they can save the post by clicking the “Save Post” button. Form validation takes place, as well as sanitation of all HTML and JavaScript entities, so users cannot input potentially harmful code into the website.

An example is shown below of an error message received when a user inputs HTML entities, and attempts to save the post:



## **Departments Listing**

Government departments can be viewed when the user navigates to the “Departments” -> “List Departments” tab on the navigational bar. These departments are stored in the database, and become available when a user creates a post (in the dropdown department list).

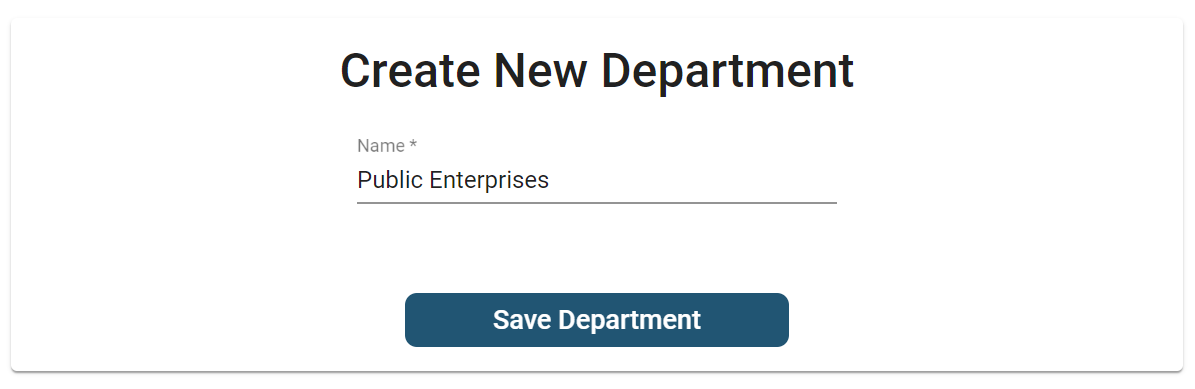


The user may create new, edit, or delete these departments in much the same way as the posts. On the following page a new government department is being created.

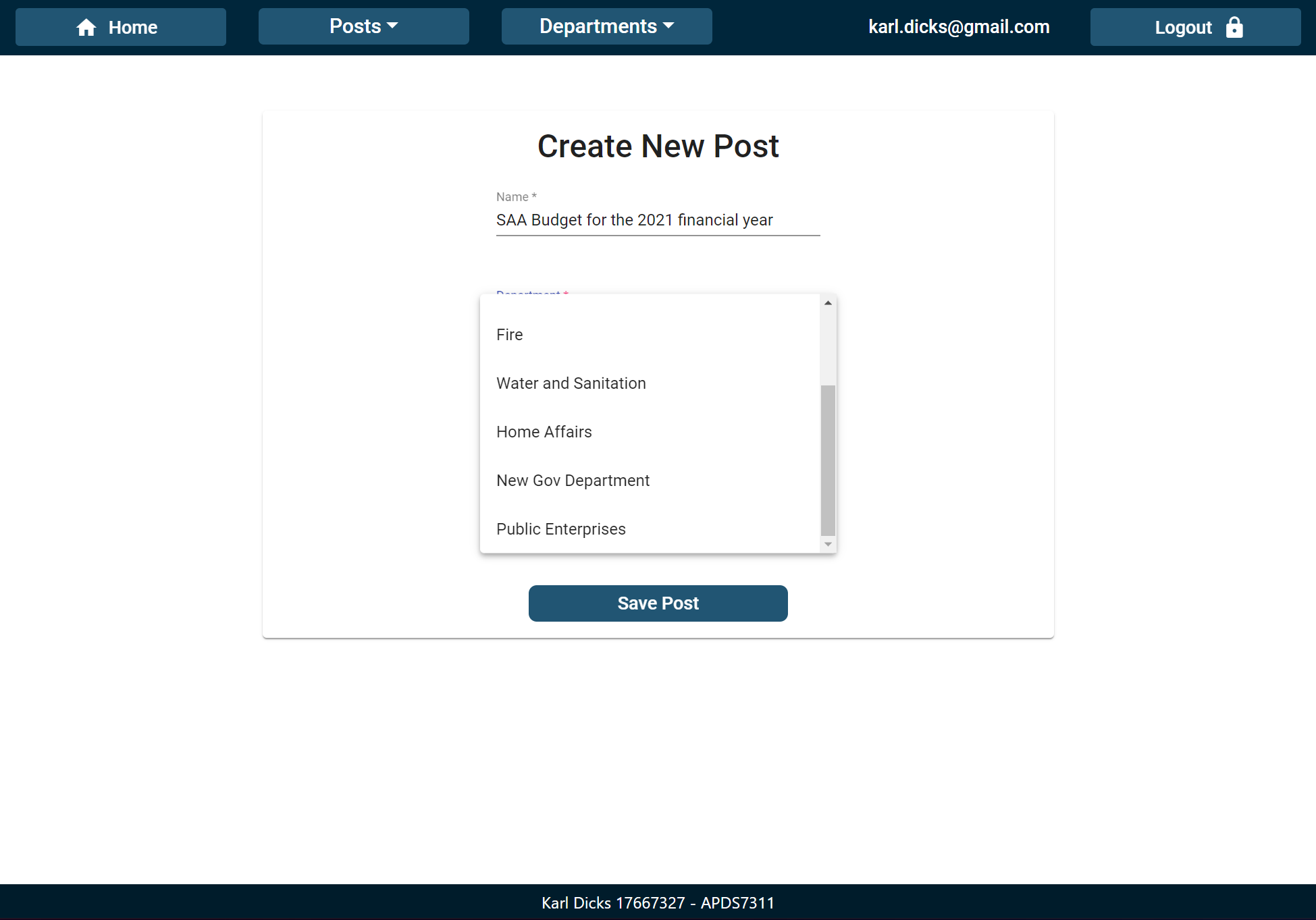
## **New Department**

New departments can be created from the “Departments” -> “New Department” page, which then allows users to create posts for these new departments. An example is provided below.

A new government department for “Public Enterprises” is being created below.



New government departments become available when creating new, or editing existing, posts.



Above is an example of a new government post being created in a new government department.

# **Conclusion**

In conclusion, the application has been developed with security in mind, with every input or output being scrutinized for potential vulnerabilities to a number of different attack methods or techniques.

From XSS injection, session jacking, brute force attacking, to username harvesting and credential theft, this application has it covered.

However, due to the constantly evolving hacking community, evolution of attack methods and techniques are inevitable, and therefore the website and web-server security will have to be assessed regularly, in order to stay up to date with prevention mechanisms, techniques, and other such methods of countering these attacks.

In addition to the above documentation, a demo video has been created and uploaded to YouTube with the following URL: <https://youtu.be/IpJkujXivr8>

With this project now in the implementation phase, we can now move on to maintaining the system, which will be done on the development environment, and then updates will be pushed to the production environment once stability and performance have been tested, to ensure that the changes have been successfully developed and applied.

To run the application please use the following commands:

1. open the project

2. run npm install in terminal

3. run npm run start:server in terminal

4. run ng serve --ssl true --ssl-cert "keys\certificate.pem" --ssl-key "keys\privatekey.pem" --proxy-config proxy.conf.json in a new terminal

I am using the online MongoDB (VC firewall may cause issues, but working at home):

Username: Karl

Password: VHl0MLhuWt2kL899

Schema: post-system

2FA has been implemented using Firebase Authentication OTP

Test Account:

Email: karl.dicks@gmail.com

Password: P@ssword123!

One Time Pin: 123456

**\*\*\* Please note the front end uses a proxy server so the command has to be:**

**ng serve --ssl true --ssl-cert "keys\certificate.pem" --ssl-key "keys\privatekey.pem" --proxy-config proxy.conf.json**

**\*\*\***

**\*\*\* Please note the application requires Git: https://git-scm.com/download/win \*\*\***

**\*\*\* Please note the application may require package-lock to be deleted in some environments before npm install can complete successfully \*\*\***

# **References**

MongoDB, n.d.. *The Mongo Shell.* [Online]   
Available at: https://docs.mongodb.com/manual/mongo/  
[Accessed 02 07 2020].

Mozilla, 2020. *Cross-Origin Resource Sharing (CORS).* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS  
[Accessed 01 07 2020].

w3resource, 2020. *Databases, documents and collections in MongoDB.* [Online]   
Available at: https://www.w3resource.com/mongodb/databases-documents-collections.php  
[Accessed 02 07 2020].