Criterion C: Development

Table of Contents

Programming languages, frameworks and libraries used

- JavaScript
- HTML
- CSS
- Bootstrap (Bootstrap, 2019)
- JQuery (The JQuery Foundation, 2019)
- Firebase (Google Developers, 2019)
- AES encryption-JS (Ricmoo, 2018)
- Js-scrypt (Garnock-Jones, 2016)

- FontAwesome (Fonticons, Inc., 2019)
- clipboard.js (Rocha, 2019)

Documents

The Solution is split between three main documents. The HTML file contains the layout and structure of the Solution. The CSS file contains the style of the solution, and essentially makes the Solution look good. The JavaScript file handles all the responsive elements of the Solution, and makes the Solution work.

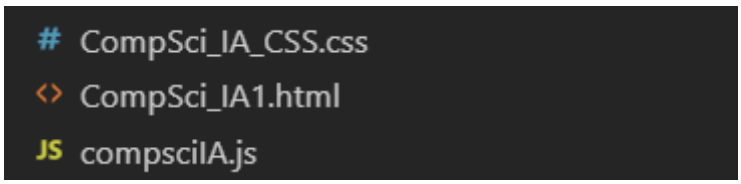


Figure 1: Documents

In addition to these documents, several other documents from outside libraries were used. These are: Bootstrap, JQuery, FontAwesome, clipboard.js, AES encryption-JS, and Js-scrypt. Furthermore, Firebase was also utilized. These are all linked at the top of the HTML file.

```
<!-- Bootstrap CSS -->
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-

<!-- Bootstrap JS -->
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5sm
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js" integrity="sha384-ZMP7rVo3mIykV+2
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+MX

<!-- Font Awesome Icons -->
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyT

<!-- Copy to clipboard -->
<script src="https://cdn.jsdelivr.net/npm/clipboard@2/dist/clipboard.min.js"></script>

<!-- Firebase-->
<script src="https://www.gstatic.com/firebasejs/5.7.2/firebase.js"></script>

<!-- AES Encryption in Javascript -->
<script type="text/javascript" src="https://cdn.rawgit.com/ricmoo/aes-js/e27b99df/index.js"></script>

<!-- Hashing JS -->
<script src="libs/scrypt.js"></script>

<!-- Own CSS-->
<link rel="stylesheet" type="text/css" href="CompSci_IA_CSS.css">
<!-- Own Javascript -->
<script src="compsciIA.js"></script>
```

Figure 2: Links at beginning of HTML

Starting Screen

The first screen that the client sees when opening the application was designed using HTML and CSS. The figures below show this screen:

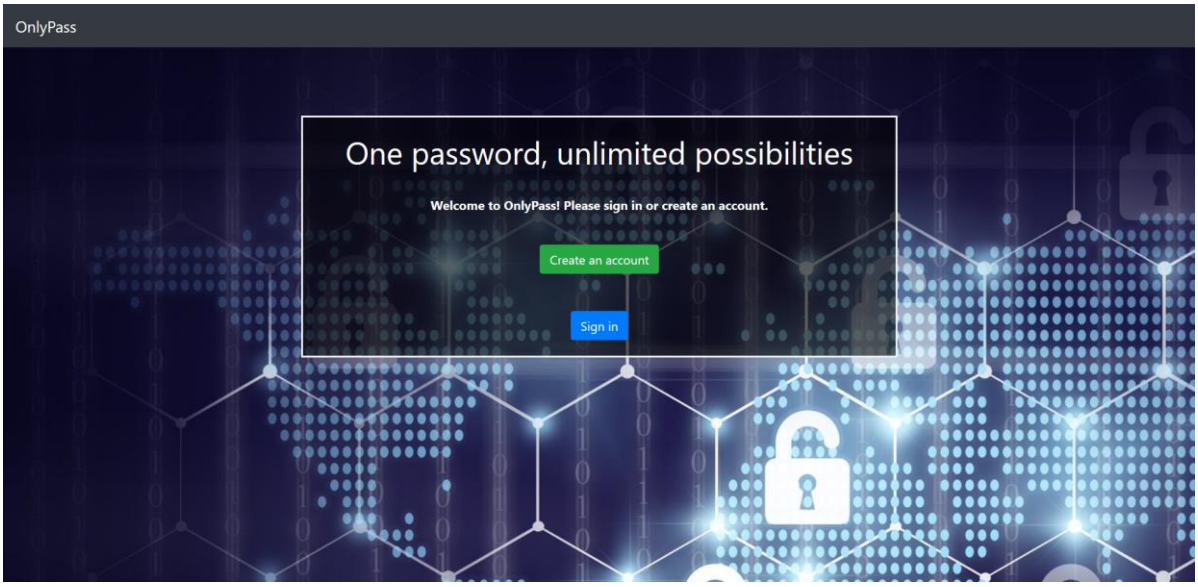


Figure 3: Starting Screen (Shutterstock, Inc., 2018)

The header at the top of this screen, where OnlyPass is displayed, was made using HTML and Bootstrap classes. The <nav> element was used, along with the classes navbar, navbar-expand-sm, bg-dark, and navbar-dark. Navbar-expand-sm is used to make the navbar mobile responsive. The other classes are used to make the navbar look the way it does.

```
<body id="body">
  <div>
  <nav class="navbar navbar-expand-sm bg-dark navbar-dark">
    <ul class="navbar-nav">
    <li class="nav-item">
      <a class="navbar-brand" href="#">OnlyPass</a>
    </li>
    </ul>
  </nav>
  </div>
```

Figure 4: Navbar HTML

The rest of the HTML code that deals with the starting screen is shown in figure 5, and the corresponding CSS in figure 6:

```
53      <div id="firstScreen" class="container">
54          <div class="row justify-content-center">
55              <div class="col-12 col-lg-6 main-window">
56                  <h1 id="heading">One password, unlimited possibilities</h1>
57                  <br>
58                  <p id="intro">Welcome to OnlyPass! Please sign in or create an account.</p>
59                  <br>
60                  <button id="register_main" class="btn btn-success" onclick="document.getElementById('register').style.display='bl
61                  <br><br>
62                  <button id="signin_main" class="btn btn-primary" onclick="openSignIn();">Sign in</button>
63              </div>
64          </div>
65      </div>
```

Figure 5: Starting screen HTML

```
.main-window {
    border: 3px solid #f1f1f1;
    background-color: rgb(0,0,0); /* Fallback color */
    background-color: rgba(0,0,0, 0.6); /* Black w/opacity/see-through */
    color: white;
    font-weight: bold;
    position: absolute;
    top: 40%;
    left: 50%;
    transform: translate(-50%, -50%);
    z-index: 2;
    width: 50%;
    padding: 20px;
    text-align: center;
}
```

Figure 6: Starting screen CSS

The relevant HTML is given the class main-window. In the CSS, this class is defined to have a white border, a black background but without 100% opacity, and to be centered, among other things. The other classes are from Bootstrap, used to make the application mobile-responsive. Trivially, some text is displayed, and there are two buttons, which when clicked open the register or sign in interfaces.

Register and sign in screens

The register and the sign in interfaces are modals, which are displayed on top of the starting page. They were created by consulting w3schools (Refsnes Data, 2019).

```css
/* The Modal (background) */
.modal {
  display: none; /* Hidden by default */
  position: fixed; /* Stay in place */
  z-index: 3; /* Sit on top */
  left: 0;
  top: 0;
  width: 100%; /* Full width */
  height: 100%; /* Full height */
  overflow: auto; /* Enable scroll if needed */
  background-color: rgb(0,0,0); /* Fallback color */
  background-color: rgba(0,0,0,0.4); /* Black w/ opacity */
  padding-top: 60px;
}
/* Modal Content/Box */
.modal-content {
  margin: 5% auto 15% auto; /* 5% from the top, 15% from the bottom and centered */
  border: 10px solid #888;
  width: 80%; /* Could be more or less, depending on screen size */
}
```

Figure 7: Modals CSS

The modals are implemented by having all HTML elements with the modal class hidden by default, and by having a higher z-index than all other elements, which results in modals being on top of everything else, once they are unhidden. The modals appear once the relevant aforementioned buttons are clicked; the second line in Figure 8 shows what code is used to make them appear. The second part of openSignIn() in Figure 8 is an event listener, and listens for the enter key being pressed (13 is the keycode for enter). If the enter key is pressed while the sign-in modal is open, the program attempts to sign the user in by running the signIn() function, which will be discussed in greater detail later. The variable signInOpen (overall line 166) is set used so that pressing the enter key will only trigger the signIn() function if the sign in screen is open. The variable is of type Boolean, and is true if the sign in screen is open, and false otherwise.

```
// -- § OPEN SIGN IN MODAL -- //

    function openSignIn(){
        document.getElementById('signin').style.display='block';
        signInOpen = true;

            // detect enter keypress
            $(document).keypress(function(e) {
                var keycode = (e.keyCode ? e.keyCode : e.which);
                if (keycode == '13') {
                    if (signInOpen === true){
                        signIn();
                    }
                }
            });
    }
```

Figure 8: openSignIn function

```
<!-- Sign in screen-->
<div class="modal" id="signin">
  <form class="modal-content animate">
    <br>
    <div class="container">
      <div class="form-group">
        <br>
        <label for="email">Email address:</label>
        <input type="email" class="form-control" id="email" required>
      </div>
      <div class="form-group">
        <label for="pwd">Password:</label>
        <input type="password" class="form-control" id="pwd" required>
      </div>
      <div class="checkbox">
        <label><input type="checkbox"> Remember me</label>
      </div>
      <div class="error" id="error_incor">Incorrect username or password</div>
      <br>
    </div>
    <br>

  <div style="background-color: #f1f1f1; padding: 16px;">
    <button type="button" onclick="closeSignIn();" class="btn btn-danger">Cancel</button>
    <button type="button" style="float:right" class="btn btn-primary" id="button_signin">Sign in</button>
  </div>

  </form>
</div>
```

Figure 9: Sign in screen HTML

```
<!-- Create an account screen-->
<div class="modal" id="register">
    <form class="modal-content animate">
        <br>
        <div class="container">
            <div class="form-group">
                <br>
                <label for="email">Email address:</label>
                <input type="email" class="form-control" id="email2" required>
            </div>
            <div class="form-group">
                <label for="pwd">Password:</label>
                <input type="password" class="form-control password-req" id="pwd2" required>
            </div>
            <div class="form-group">
                <label for="pwd">Confirm password:</label>
                <input type="password" class="form-control" id="cpwd" required>
            </div>
            <div class="checkbox">
                <label><input type="checkbox" id="terms"> I have read the <a href="onlypass_terms.pdf" target="_blank">Terms a
            </div>
            <div class="error" id="error_match">Passwords don't match</div>
            <div class="error" id="error_short">Password is too short</div>
            <div class="error" id="error_number">Password must include a number</div>
            <div class="error" id="error_letter">Password must include a letter</div>
            <div class="error" id="error_terms">You must accept the Terms and Conditions</div>
            <br>
        </div>
        <br>
        <div style="background-color: #f1f1f1; padding: 16px;">
            <button type="button" onclick="closeRegister();" class="btn btn-danger">Cancel</button>
            <button type="submit" style="float:right" class="btn btn-success" id="button_register">Create an account</button>
        </div>
    </form>
</div>
```

Figure 10: Register screen HTML

Figures 9 and 10 show the HTML of the sign in and register screens, respectively. Both contain a form, which the user fills out. Both also contain errors, which are hidden by default by giving them the class error, which is defined in CSS to cause elements to be hidden by default. Furthermore, the class also causes elements to appear in red, to draw the user's attention to the error. Both screens contain two buttons, one to close the modal and one to sign in or register. The button clicks are handled directly (using onclick) for the cancel buttons, and Figure 11 shows the event listeners which handle the sign in and register buttons. Since the application is a single-page application, these are used to prevent the form from being submitted, thus preventing the page from being reloaded.

```
// -- § MAKE THE APPLICATION SINGLE-PAGE -- //

    //Make it so the form isn't submitted when the Sign In button is clicked, but signIn() is run instead
    $(document).ready(function() {
        $("#button_signin").on("click", function(ev){
            console.log("login clicked");
            ev.preventDefault();
            signIn();
        });
    });

    //Make it so the form isn't submitted when the Create an Account button is clicked, but createUser() is run instead
    $(document).ready(function() {
        $("#button_register").on("click", function(ev){
            console.log("register clicked");
            ev.preventDefault();
            createUser();
        });
    });
```

Figure 11: Event listeners for sign in and register buttons

Once the createUser() function is triggered, first all errors currently displayed will be hidden, so that there is a clean slate. Next, it will be checked whether the password and confirm password entries match. Moreover, the password will be checked against several criteria, namely whether it is sufficiently long and whether it contains both letters and numbers. Finally, it will be checked whether the user has accepted the Terms and Conditions. Should any of these checks fail, the relevant error message will be displayed and the function will stop execution. Otherwise, the function will attempt to create a user.

```
86  // -- § CREATE A USER -- //
87
88      function createUser(){
89          hideErrorsOnRegister();
90          var email = $("#email2").val();
91          var pwd = $('#pwd2').val();
92          if (pwd !== $('#cpwd').val()){
93              $("#error_match").css("display","block")
94              //alert("Passwords don't match!");
95              return;
96          } else if (pwd.length < 8) {
97              $("#error_short").css("display","block")
98              //alert("Password is too short!");
99              return;
100         } else if (pwd.match(/\d/) === null){
101             $("#error_number").css("display","block")
102             //alert("Password must include a number!");
103             return;
104         } else if (pwd.match(/\D/) === null) {
105             $("#error_letter").css("display","block")
106             //alert("Password must include a letter");
107             return;
108         } else if ($('#terms').is(':checked') === false) {
109             $("#error_terms").css("display","block")
110             //alert("You must accept the Terms and Conditions!");
111             return;
112         }
113
114         console.log("trying to create user");
115         newUser = true;
116
117         firebase.auth().createUserWithEmailAndPassword(email, pwd).catch(function(error) {
118         // Handle Errors here.
119
120         var errorCode = error.cod any
121         var errorMessage = error.message;
122         console.log("error creating user");
123         alert(errorMessage);
124         newUser = false;
125         //ADD SUCCESS MESSAGE
126         return;
127         // ...
128
129         });
130     }
```

Figure 12: createUser function

User authentication is done via Firebase. This framework makes it possible to use predefined functions (e.g. line 80) to create user accounts and authenticate users. Firebase also serves the purpose of storing data.

Firebase and user authentication

Firebase first has to be initialized, which is shown in the figure below. A link to the relevant Firebase project is established, and a connection with the online database is created. The fact that the database is online makes it easier to sync passwords across devices. The potential drawback of unsecure storage is removed by only sending already encrypted data to Firebase. The encryption key is never sent.

```
13      // Initialize Firebase
14      var config = {
15          apiKey: "AIzaSyBt71Qd9Y9B56iua99Khs41NQxthxuysXc",
16          authDomain: "compsciia-f176a.firebaseapp.com",
17          databaseURL: "https://compsciia-f176a.firebaseio.com",
18          projectId: "compsciia-f176a",
19          storageBucket: "compsciia-f176a.appspot.com",
20          messagingSenderId: "567193459229"
21      };
22      firebase.initializeApp(config);
23
24      // Initialize Cloud Firestore through Firebase
25      var db = firebase.firestore();
26
27      // Disable deprecated features of Firebase
28      db.settings({
29          timestampsInSnapshots: true
30      });
```

Figure 13: Initialization of Firebase

Once a new user account is created, the user has to sign in. This is done through the sign in screen as previously discussed, and the signIn() function is run.

```
133  // -- § SIGN IN AN EXISTING USER -- //
134
135      //Sign in an existing user
136      function signIn(){
137          signInOpen = false;
138          $("#error_incor").css('display','none');
139          var email = $("#email").val();
140          userPassword = $("#pwd").val();
141          console.log("trying to log in",email, userPassword);
142          firebase.auth().signInWithEmailAndPassword(email, userPassword).catch(function(error) {
143              // Handle Errors here.
144              //alert("Incorrect username or password");
145              signInOpen = true;
146              $("#error_incor").css('display','block');
147              var errorCode = error.code;
148              var errorMessage = "This is an error";
149
150              // ...
151          });
152      }
```

Figure 14: signIn() function

A potential error message is first hidden, and then another predefined Firebase function is run. If there is no matching email and password pair, an error is displayed. Otherwise, the user is authenticated. There is an event listener that listens for a change of the authentication state, shown in Figures 15 and 16 below:

```
372  // -- § RUN UPON SIGN-IN -- //
373
374      firebase.auth().onAuthStateChanged(function(user) {
375          if (user) {
376              if (newUser === false){
377              console.log("user signed in");
378              if (user.emailVerified !== true){
379                  signOut();
380                  alert("You need to verify your account via email before signing in.")
381                  return;
382              }
383
384              document.getElementById("signin").style.display='none';
385              document.getElementById("firstScreen").style.display='none';
386              document.getElementById("intro").style.display='none';
387              $(".password-list-screen").css("display", "inline");
388              $('body').css('background-image', '');
389
390              email = user.email;
391
392              var user = firebase.auth().currentUser;
393              userId = user.uid;
394              console.log(userId);
```

Figure 15: Authentication listener (part 1)

Once the authentication listener detects that a user has signed in, all elements of the starting screen are hidden, and the main interface of the program is displayed.

```
411             var docRef = db.collection("users").doc(userId);
412
413             docRef.get().then(function(doc) {
414                 if (doc.exists) {
415                     console.log("Document data:", doc.data());
416                     var docData = doc.data();
417                     console.log(docData);
418                     decrypt(docData.encrypted_data);
419
420                     //For all password entries, display those
421                     listPasswords();
422
423                     var clipboard = new ClipboardJS('.copy');
424
425                     $("#current_date").html(currentDate());
```

Figure 16: Authentication listener (part 2)

The program then obtains the encrypted password list from Firebase, and runs the function to decrypt and parse it (see decryption section). Next, the listPasswords() function is run, in which a loop displays all the different passwords for different websites in an orderly table (Figures 17, 18).

```
261     //Display all the passwords from the passwords array in a table
262     function listPasswords(){
263         for (i = 0; i < passwords.length; i++){
264             var table = document.getElementById('password_list');
265             var row = table.insertRow(rowNumber);
266             row.id = "row" + rowNumber;
267             rowNumber++;
268             var cell0 = row.insertCell(0);
269             var cell1 = row.insertCell(1);
270             var cell2 = row.insertCell(2);
271             var cell3 = row.insertCell(3);
272             var cell4 = row.insertCell(4);
273             var cell5 = row.insertCell(5);
274             var cell6 = row.insertCell(6);
275             var cell7 = row.insertCell(7);
276             var cell8 = row.insertCell(8);
277             cell0.innerHTML = "<span class='editable'>"+passwords[i].name+"</span>";
278             cell1.innerHTML = "<span class='editable'>"+passwords[i].website+"</span>";
279             cell2.innerHTML = "<span>"+passwords[i].dlc+"</span>";
280             cell3.innerHTML = "<span class='editable password'>"+passwords[i].password+"</span>";
281             cell4.innerHTML = "<button class='btn btn-light eye'><i class='fas fa-eye'></i></button>";
282             cell5.innerHTML = "<button class='btn btn-light copy' data-clipboard-text='" + passwords[i].password + "'><i class='f
283             cell6.innerHTML = buttonGenerate;
284             cell7.innerHTML = buttonEdit;
285             cell8.innerHTML = buttonRemove;
286         }
287     }
```

Figure 17: listPasswords() function

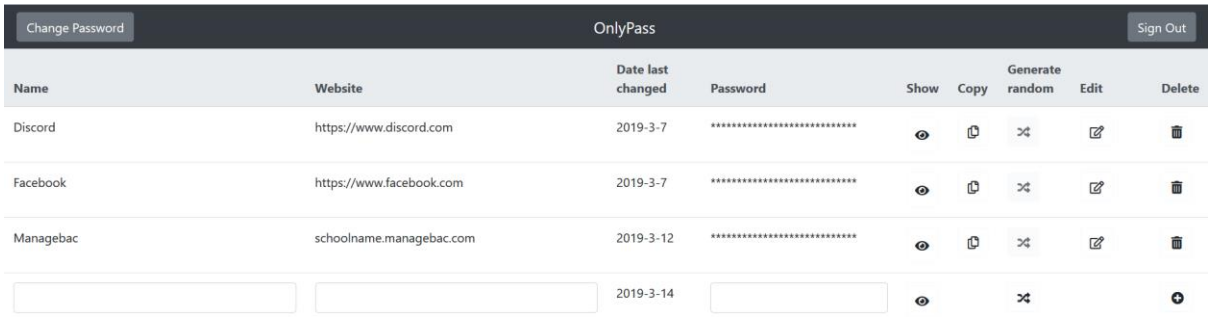


Figure 18: Password list screen

Making changes to the password list

Adding an entry

```
310     //Function to add a password
311     function addEntry(name, website, password){
312
313         //backend --> adds the new entry to the array that is encrypted and stored
314         var entry = new passwordEntry(name, website, password);
315         passwords.push(entry);
316         console.log(passwords);
317         encrypt();
318
319         //frontend --> adds the new entry to the display
320         var table = document.getElementById('password_list');
321         var row = table.insertRow(rowNumber);
322         rowNumber++;
323         var cell0 = row.insertCell(0);
324         var cell1 = row.insertCell(1);
325         var cell2 = row.insertCell(2);
326         var cell3 = row.insertCell(3);
327         var cell4 = row.insertCell(4);
328         var cell5 = row.insertCell(5);
329         var cell6 = row.insertCell(6);
330         var cell7 = row.insertCell(7);
331         var cell8 = row.insertCell(8);
332         cell0.innerHTML = "<span class='editable'>"+name+"</span>";
333         cell1.innerHTML = "<span class='editable'>"+website+"</span>";
334         cell2.innerHTML = currentDate();
335         cell3.innerHTML = "<span class='editable password'>"+password+"</span>";
336         cell4.innerHTML = "<button class='btn btn-light eye'><i class='fas fa-eye'></i></button>";
337         cell5.innerHTML = "<button class='btn btn-light copy' data-clipboard-text='" + password + "'><i class='far fa-copy'></i><
338         cell6.innerHTML = buttonGenerate;
339         cell7.innerHTML = buttonEdit;
340         cell8.innerHTML = buttonRemove;
341     }
```

Figure 19: addEntry() function

To add a new password, the addEntry() function is run. This creates a new instance of the passwordEntry class and pushes it to the passwords array, then calls the encrypt() function to save changes. Next, the new password is displayed at the bottom of the table.

Removing an entry

```
535     //Open are you sure screen when deleting passwords
536     $("body").on("click", ".delete", function(){
537         rowBeingDeleted = $(this).get(0);
538         console.log(rowBeingDeleted);
539         console.log($(this));
540         $("#delete_password").css("display", "block");
541     })
```

Figure 20: Deleting a password

Once the user clicks the delete icon, a modal opens, prompting a confirmation.

```
166 <!-- Modal for deleting passwords -->
167 <div class="modal" id="delete_password">
168     <form class="modal-content animate sure">
169         <div class="container">
170             <br>
171             Are you sure you wish to delete your login details for this website? Once deleted, login details cannot be recovered.
172             <br><br>
173         </div>
174         <br>
175         <div style="background-color:#f1f1f1; padding: 16px;">
176         <button type="button" onclick="closeAreSure('delete_password');" class="btn btn-primary">Cancel</button>
177         <button type="button" style="float:right" class="btn btn-danger" id="button_confirm_delete" onclick="deleteEntry(rowBeingDeleted); closeAreSure('delete_passwor
178         </div>
179     </form>
180 </div>
```

Figure 21: Modal for deleting passwords HTML

Should this be given, deleteEntry() is run.

```
343     //Function to delete a password
344     function deleteEntry(r){
345         var i = r.parentNode.parentNode.rowIndex;
346         passwords.splice(i-1,1);
347         document.getElementById('password_table').deleteRow(i);
348         rowNumber--;
349         encrypt();
350     }
```

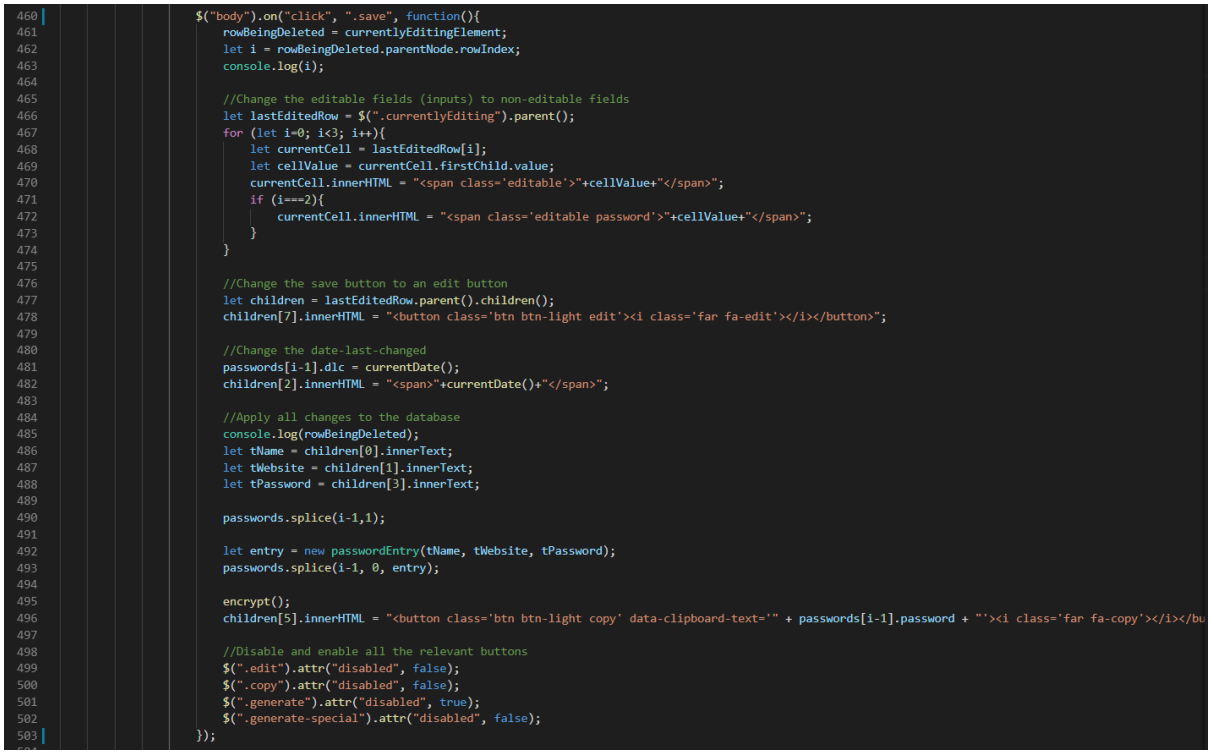
Figure 22: deleteEntry() function

Editing an entry

```
//Making the rows editable, one at a time
var rowEditing = false;
var currentlyEditingElement = null;
$("body").on("click", ".edit", function(){
    if (rowEditing === true){
        let lastEditedRow = $(".currentlyEditing").parent();
        console.log(lastEditedRow);

    }
    console.log("Attemting to make row editable");
    let editableRow = $(this).parent().parent();
    let rowId = editableRow.attr("id");
    let children = editableRow.children();
    for (let i=0; i<4; i++){
        if (i !== 2){
        let currentChild = children[i];
        let childValue = currentChild.innerText
        currentChild.innerHTML = "<input class='form-control currentlyEditing' value="+childValue+">";
            if (i===3){
                currentChild.innerHTML = "<input type='password' class='form-control password-editing currently
            }
        }
    }
    currentlyEditingElement = children[7];
    children[7].innerHTML = "<button class='btn btn-light save'><i class='far fa-save'></i></button><button cla
    children[6].innerHTML = "<button class='btn btn-light generate'><i class='fas fa-random'></i></button>";

    $(".edit").attr("disabled", true);
    $(".copy").attr("disabled", true);
    rowEditing = true;
    console.log(editableRow);
});
```

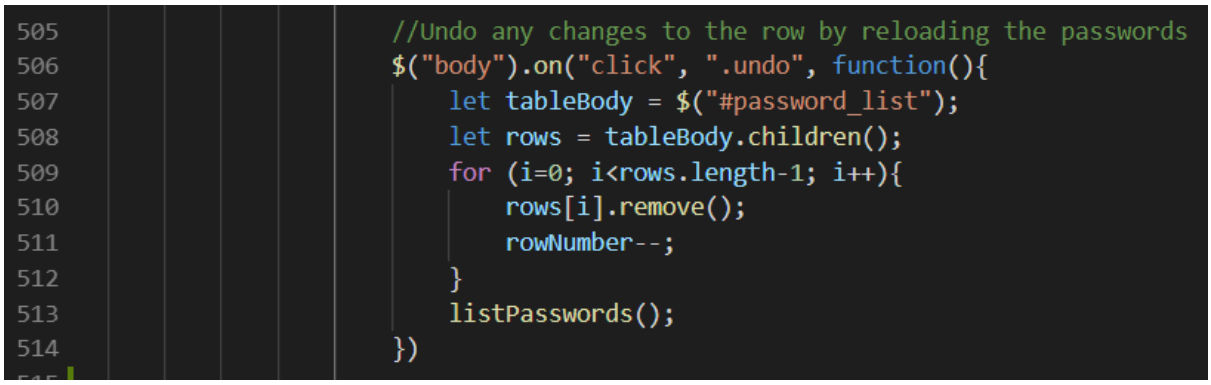
Figure 23: Editing an entry

Once the edit icon is clicked, the relevant row is obtained and elements are cycled through in a loop, where their innerHTML is changed to turn them into input fields. Save and undo buttons replace the edit button. Figures 24 and 25 show how changes are saved or undone.



```
$("body").on("click", ".save", function(){
    rowBeingDeleted = currentlyEditingElement;
    let i = rowBeingDeleted.parentNode.rowIndex;
    console.log(i);

    //Change the editable fields (inputs) to non-editable fields
    let lastEditedRow = $(".currentlyEditing").parent();
    for (let i=0; i<3; i++){
        let currentCell = lastEditedRow[i];
        let cellValue = currentCell.firstChild.value;
        currentCell.innerHTML = "<span class='editable'>"+cellValue+"</span>";
        if (i===2){
            currentCell.innerHTML = "<span class='editable password'>"+cellValue+"</span>";
        }
    }

    //Change the save button to an edit button
    let children = lastEditedRow.parent().children();
    children[7].innerHTML = "<button class='btn btn-light edit'><i class='far fa-edit'></i></button>";

    //Change the date-last-changed
    passwords[i-1].dlc = currentDate();
    children[2].innerHTML = "<span>"+currentDate()+"</span>";

    //Apply all changes to the database
    console.log(rowBeingDeleted);
    let tName = children[0].innerText;
    let tWebsite = children[1].innerText;
    let tPassword = children[3].innerText;

    passwords.splice(i-1,1);

    let entry = new passwordEntry(tName, tWebsite, tPassword);
    passwords.splice(i-1, 0, entry);

    encrypt();
    children[5].innerHTML = "<button class='btn btn-light copy' data-clipboard-text='" + passwords[i-1].password + "'><i class='far fa-copy'></i></bu

    //Disable and enable all the relevant buttons
    $(".edit").attr("disabled", false);
    $(".copy").attr("disabled", false);
    $(".generate").attr("disabled", true);
    $(".generate-special").attr("disabled", false);
});
```

Figure 24: Save changes



```
//Undo any changes to the row by reloading the passwords
$("body").on("click", ".undo", function(){
    let tableBody = $("#password_list");
    let rows = tableBody.children();
    for (i=0; i<rows.length-1; i++){
        rows[i].remove();
        rowNumber--;
    }
    listPasswords();
})
```

Figure 25: Undo changes

Encryption

AES-256 encryption is used. This is one of the most secure forms of encryption available and is therefore used as the client specifically wished for the encryption to be robust and as secure as possible.

Hashing

```
654  // -- § GET AN ENCRYPTION KEY FROM USER PASSWORD -- //
655
656      //New scrypt: https://github.com/tonyg/js-scrypt
657      function getKey(password){
658          let returnVal = null;
659          scrypt_module_factory(function (scrypt) {
660              var keyBytes = scrypt.crypto_scrypt(scrypt.encode_utf8(password), scrypt.encode_utf8(userEmail), 16384, 8, 1, 16)
661              returnVal = keyBytes;
662          });
663          return returnVal;
664      }
```

Figure 26: Hashing

The user's password is hashed to obtain an encryption key using Js-scrypt (Garnock-Jones, 2016).

Decrypting

```
712
713      // Decrypts into plaintext
714      function decrypt(input){
715          var key = getKey(userPassword);
716          var encryptedBytes = aesjs.utils.hex.toBytes(input);
717
718          // The counter mode of operation maintains internal state, so to
719          // decrypt a new instance must be instantiated.
720          var aesCtr = new aesjs.ModeOfOperation.ctr(key, new aesjs.Counter(5));
721          var decryptedBytes = aesCtr.decrypt(encryptedBytes);
722
723          // Convert our bytes back into text
724          var decryptedText = aesjs.utils.utf8.fromBytes(decryptedBytes);
725
726          // Convert decrypted string to an array with password objects and make the passwords array equal to it
727          passwords = JSON.parse(decryptedText);
728      }
```

Figure 27: decrypt() function

The decryption function, run when the user logs into the application, first calls the hashing function to obtain a key, uses it to decrypt the passwords, before ultimately setting the passwords array equal to a parsed version.

Encrypting

```
669     // Encrypts the plaintext
670     function encrypt() {
671
672         // Get an encryption key
673         var key = getKey(userPassword);
674
675         // Convert password objects in the array to a string
676         var text = JSON.stringify(passwords);
677
678         // Convert text to bytes
679         var textBytes = aesjs.utils.utf8.toBytes(text);
680
681         // The counter is optional, and if omitted will begin at 1
682         var aesCtr = new aesjs.ModeOfOperation.ctr(key, new aesjs.Counter(5));
683         var encryptedBytes = aesCtr.encrypt(textBytes);
684
685         // To print or store the binary data, you may convert it to hex
686         var encryptedHex = aesjs.utils.hex.fromBytes(encryptedBytes);
687         console.log(encryptedHex);
688         // "a338eda3874ed884b6199150d36f49988c90f5c47fe7792b0cf8c7f77eeffd87
689         //  ea145b73e82aefcf2076f881c88879e4e25b1d7b24ba2788"
690         //document.getElementById('a').innerHTML=encryptedHex;
691
692         var docRef = db.collection("users").doc(userId);
693         var setWithMerge = docRef.set({
694             encrypted_data: encryptedHex
695         }, { merge: true });
```

Figure 28: encrypt() function

Encrypt() likewise calls getKey(). It then converts the array of password objects to a string of bits, encrypts it, and saves it to Firebase.

Word Count: 1223

References

Bootstrap. (2019). *Bootstrap*. Retrieved from https://getbootstrap.com

Fonticons, Inc. (2019). *Font Awesome*. Retrieved from https://fontawesome.com/

Garnock-Jones, T. (2016). *Js-scrypt: Pure-Javascript Emscripten-compiled scrypt routine*. Retrieved from https://github.io/tonyg/js-scrypt/

Google Developers. (2019). *Firebase*. Retrieved from https://firebase.google.com/

Refsnes Data. (2019). *How to - Login Form*. Retrieved from w3schools: https://www.w3schools.com/howto/howto_css_login_form.asp

Ricmoo. (2018). *AES-JS*. Retrieved from https://github.com/ricmoo/aes-js

Rocha, Z. (2019). *clipboard.js*. Retrieved from https://clipboardjs.com/

Shutterstock, Inc. (2018). *Shutterstock*. Retrieved from https://www.shutterstock.com

The JQuery Foundation. (2019). *jQuery CDN – Latest Stable Versions*. Retrieved from https://code.jquery.com/