

MyPass Report

CIS 476 Term Project

Group: Shahd Mustafa and Zeinab Alrubaiee

Table Of Contents

Project description:.....	3
Class Diagrams:.....	3
- Singleton Pattern:.....	3
- Chain of Responsibility Pattern:.....	5
- Builder Pattern:.....	8
- Proxy Pattern:.....	13
Database Schema:.....	14
User Interface:.....	14
References:.....	19

Team tasks:

- Singleton pattern and chain of responsibility pattern (code and report description): Shahd Mustafa
- Proxy and builder pattern (code and report description): Zeinab Alrubaiee
- Both members worked on the html/css and database of the code.

Project description:

MyPass software application stores and secures users' sensitive information such as license number, passport number, and credit card information. The site allows users to create an account using a username, email, password, and store answers to the three security questions. Users are able to create their own password or use the password generator for a strong and secure password. Users are also able to recover forgotten passwords by answering the three security questions correctly. Once users are logged in to MyPass they will have access to view and edit all their sensitive information. To ensure that the information is secured the site masks the data until the user wishes to view it.

Site link: <https://shahdmu34.github.io/TermProject-476/>

Class Diagrams:

- Singleton Pattern:

The singleton pattern is used to manage the user login session on the site, ensuring that only one instance is logged in at a time.



The class diagram illustrates the functions used in the implementation of the singleton pattern:

- constructor - this method checks if there is a preexisting instance already, if an instance exists the program returns the instance, if there is no instance then it assigns a new instance using database and auth.

```

class LoginAuthPattern {
  constructor() {
    if (LoginAuthPattern.instance) {
      return LoginAuthPattern.instance;
    }

    this.auth = getAuth(app);
    this.database = getDatabase(app);

    LoginAuthPattern.instance = this;
  }
}

```

- `signInWithEmailAndPassword` - this method takes in two parameters that are used to sign the user in. This method utilizes the Firebase Auth to authenticate the user from the provided email and password. If the authentication is a success, the database is updated with the login date and redirects the user to the home page of the site. If the email or password is incorrect, an error message is displayed.

```

signInWithEmailAndPassword(email, password) {
  return signInWithEmailAndPassword(this.auth, email, password)
    .then((userCredential) => {
      const user = userCredential.user;
      const past_login = new Date();

      update(ref(this.database, 'Users/' + user.uid), {
        pastlogin: past_login,
      })
        .then(() => {
          alert('User ' + email + ' is logged in!');
          window.location.href = "home.html";
        })
        .catch((error) => {
          alert('Error updating user: ' + error.message);
        });
    })
    .catch((error) => {
      const errorMessage = error.message;
      alert(errorMessage);
    });
}
}

```

- `submitBtn.addEventListener` - this event listener handles the user interaction with the login button. In this method, the inputted user information email and

password are retrieved and the LoginAuthInstance is created and calls on the signInWithEmailAndPassword to check if the user exists or not.

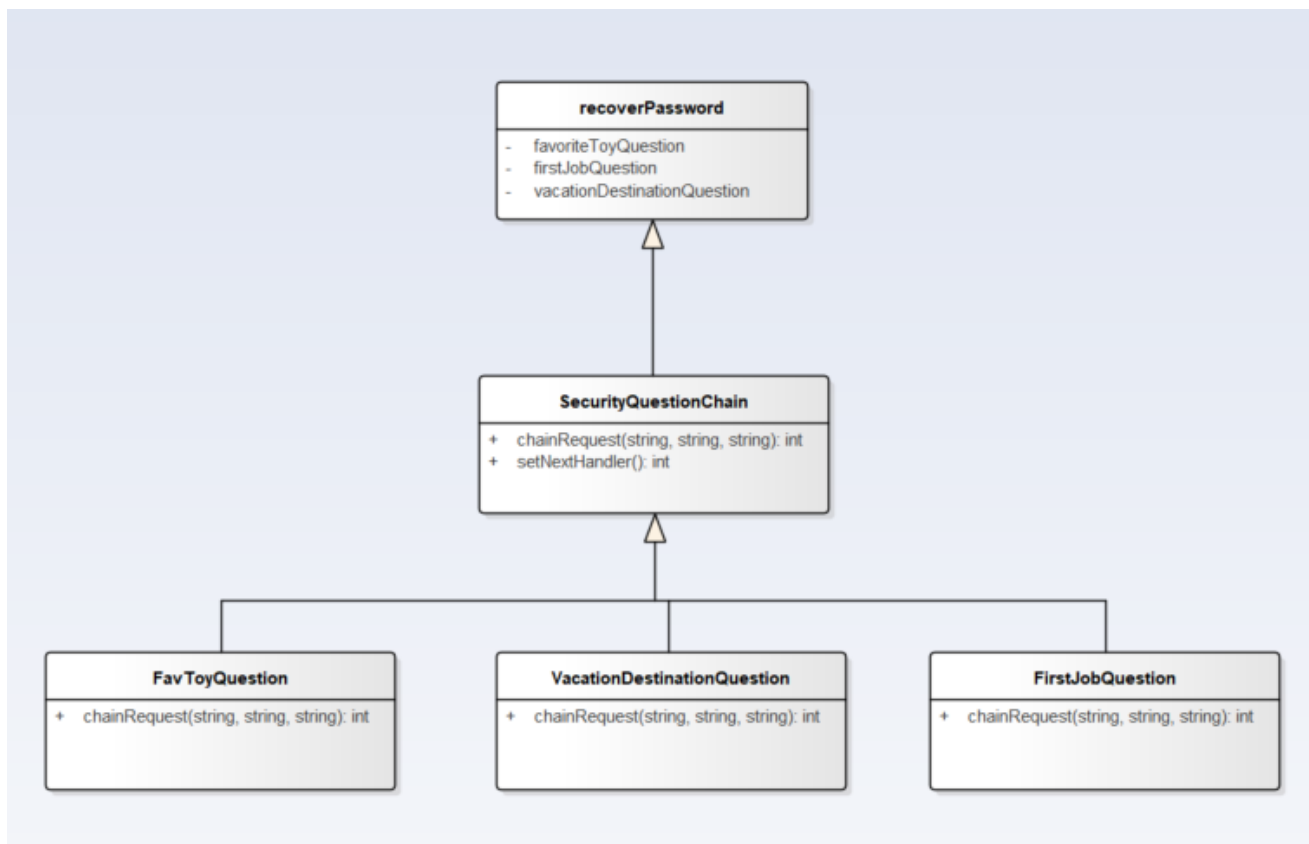
```
const LoginAuthInstance= new LoginAuthPattern();

submitBtn.addEventListener('click', (e) => {
const email = document.getElementById('email').value;
const password = document.getElementById('password').value;

LoginAuthInstance.signInWithEmailAndPassword(email, password);
});
```

- Chain of Responsibility Pattern:

The chain of responsibility pattern was used in the pass.html file to create a secure process for users to recover a forgotten password by inputting their username, email, and answering three security questions. The pattern was used for handling the three questions by creating a chain that checks if the inputted user answer matches the stored answers for the account, if the answer is valid it then checks the next. After the third question is processed and all answers are valid the program will display the master password for the user in the alert, however, if one answer is invalid the program will not display the password and the user will need to try again by entering valid data.



The class diagram above explains how the classes interact with each other and the information that each one handles, which then interacts with the submit button that gives the result.

- **recoverPassword** - class/function is responsible for switching between questions and creating the chain. If all questions are answered correctly the master_password is retrieved from the database and is displayed to the user. If questions are not answered correctly, an alert is sent stating "Security questions are not valid." Both paths will then end the program, leading to the login page.

```
function recoverPassword(email, answer1, answer2, answer3, userData) {
    // Create and Link the chain of responsibility
    const favoriteToyQuestion = new FavToyQuestion();
    const vacationDestinationQuestion = new VacationDestinationQuestion();
    const firstJobQuestion = new FirstJobQuestion();

    favoriteToyQuestion.setNextHandler(vacationDestinationQuestion);
    vacationDestinationQuestion.setNextHandler(firstJobQuestion);

    // Trigger the chain with the user's answers and userData
    const isValidQuestion1 = favoriteToyQuestion.chainRequest("Favorite Childhood Toy or Game", answer1, userData);
    const isValidQuestion2 = vacationDestinationQuestion.chainRequest("Dream Vacation Destination", answer2, userData);
    const isValidQuestion3 = firstJobQuestion.chainRequest("First Job or Company", answer3, userData);

    // retrieving the master password from the database
    const masterPassword = userData.master_password;

    // Check if all three questions are valid
    if (isValidQuestion1 && isValidQuestion2 && isValidQuestion3) {
        alert('User Master Password: ' + masterPassword);
    } else {
        alert('Security questions are not valid');
    }
}
```

- SecurityQuestionChain - this class is used to handle the next request in the chain pattern. The setNextHandler handles passing the request to the next chain. chainRequest takes in three parameters: question (security question being checked), inputAnswer (user answer), and storedAnswer (answer stored in the database). This function checks if there is a next handler to pass on next. If there isn't it returns false indicating the chain has ended. This class is triggered when the user submits their answers to the security questions.

```
class SecurityQuestionChain {
    setNextHandler(handler) {
        this.nextHandler = handler;
    }

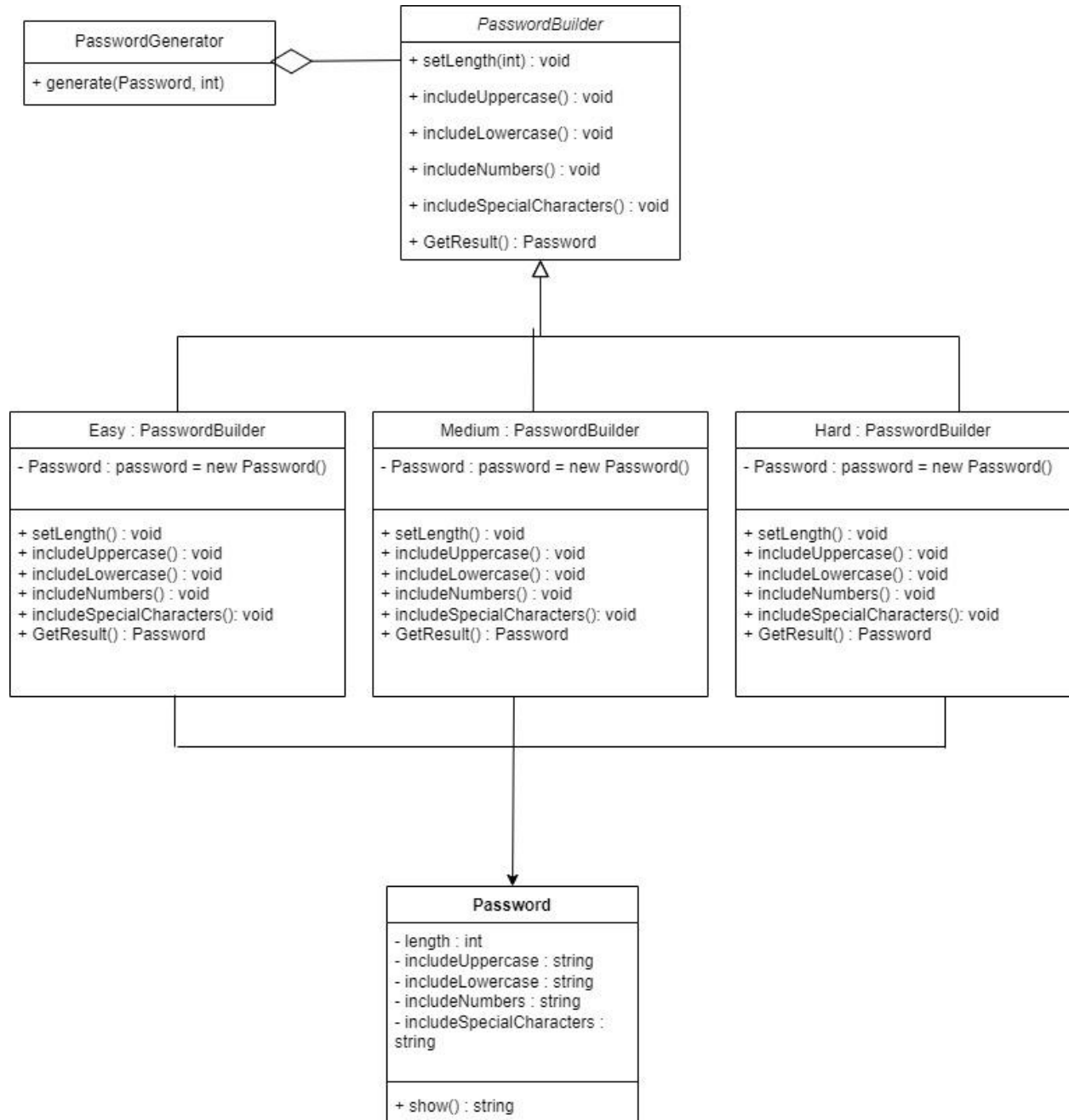
    chainRequest(question, inputAnswer, storedAnswer) {
        if (this.nextHandler) {
            return this.nextHandler.chainRequest(question, inputAnswer, storedAnswer);
        } else {
            return false; // End of the chain
        }
    }
}
```

- FavToyQuestion, VacationDestinationQuestion, FirstJobQuestion - these three classes are responsible for checking the inputted answers for each security question. The

method `chainRequest` takes in three parameters: `question` (security question being checked), `inputAnswer` (user inputted answer), `userData` (stored answer). The method checks if there is any `userData` stored for the question. If it is true, then it checks if the inputted answer matches the stored answer. If both conditions are true, then the answer is valid and the chain continues to check the next. But if either one is false, then it continues checking the next question but returns false (the recover password will handle the outputted answer for both false and true answers).

```
// Question one
class FavToyQuestion extends SecurityQuestionChain {
  chainRequest(question, inputAnswer, userData) {
    if (userData && userData.question_one) {
      const storedAnswer = userData.question_one.trim();
      return inputAnswer.trim() === storedAnswer;
    } else {
      this.chainRequest(question, inputAnswer, userData);
      return false;
    }
  }
}
```

- `submitBtn.addEventListener` - this button handler is what connects the chain of responsibility pattern to the HTML and the database. When the user has finished inputting all their answers and interacts with the submit button, the input fields for username, email, question1, question2, and question3 are retrieved. The method checks if the inputted username and email are in the database; if so, it calls on the `recoverPassword` function, which will initiate the start of the chain pattern, thus checking if the answers are valid and outputting the master password. After the alert, it switches back to the login page or outputs an error message.
- **Builder Pattern:**
- The Builder Pattern's purpose is to provide an expressway to deviate from the three varying difficulties of password generation: easy, medium, and hard. It does so by combining a series of different classes, one a builder, another a concrete builder, a director, and a product. All these cooperate together in order to successfully generate three different types of passwords, all according to the user's preferences.



- The above diagram illustrates the classes that were implemented into the builder pattern. In this instance, the PasswordGenerator is the Director, PasswordBuilder is the Builder, Easy, Medium, and Hard are the ConcreteBuilders, and Password is the Product.

```
//Director
1 usage
class PasswordGenerator {
  1 usage
  generate(passwordInstance, length) {
    let pass = new Hard();
    pass = passwordInstance;
    pass.setLength(length);
    pass.includeUppercase();
    pass.includeLowercase();
    pass.includeNumbers();
    pass.includeSpecialCharacters();
  }
}
```

- It all begins with the client class, in this case being PasswordGenerator. The client receives a call from the event listener and chooses between easy, medium, and hard depending on what the user desires, as well as incorporating a user specified length.

```

//Builder
3 inheritors 3 usages
class PasswordBuilder {
    3 overrides 1 usage
    setLength(length) {

    }

    3 overrides 1 usage
    includeUppercase() {

    }

    3 overrides 1 usage
    includeLowercase() {

    }

    3 overrides 1 usage
    includeNumbers() {

    }

    3 overrides 1 usage
    includeSpecialCharacters() {

    }

    3 overrides 1 usage
    GetResult() {}
}

```

- It then moves onto the Builder class, appropriately called PasswordBuilder. This is responsible for constructing each part of the password depending on its difficulty.
- It then transitions into the Concrete Builders, which gathers and assembles the parts of the desired product by utilizing the Builder interface. This is broken down into three different classes: Easy, Medium, and Hard, and each class extends PasswordBuilder. The differences between these are minimal, but involve the following:
 - Easy includes only uppercase and lowercase letters.
 - Medium includes uppercase letters, lowercase letters, and numbers.
 - Hard includes all the above as well as special characters.
 - An example of one of the classes is shown below:

```

//Concrete Builders
//Easy only incorporates uppercase and lower case letters
1 usage
class Easy extends PasswordBuilder {
    password = new Password;

    no usages
    setLength(length) {
        this.password.length = length;
    }

    no usages
    includeUppercase() {
        this.password.includeUppercase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    }

    no usages
    includeLowercase() {
        this.password.includeLowercase = "abcdefghijklmnopqrstuvwxyz";
    }

    no usages
    includeNumbers() {
        this.password.includeNumbers = "";
    }

    no usages
    includeSpecialCharacters() {
        this.password.includeSpecialCharacters = "";
    }

    no usages
    GetResult() {
        return this.password;
    }
}

```

- Lastly is the Product, being the Password class itself. This encapsulates the object that is being constructed and is built and defined by the ConcreteBuilder. It also combines every element of the Password and returns a string object, resulting in a password that has been generated from this combination of classes.

```

// Product
3 usages
class Password {
  length;
  includeUppercase;
  includeLowercase;
  includeNumbers;
  includeSpecialCharacters;

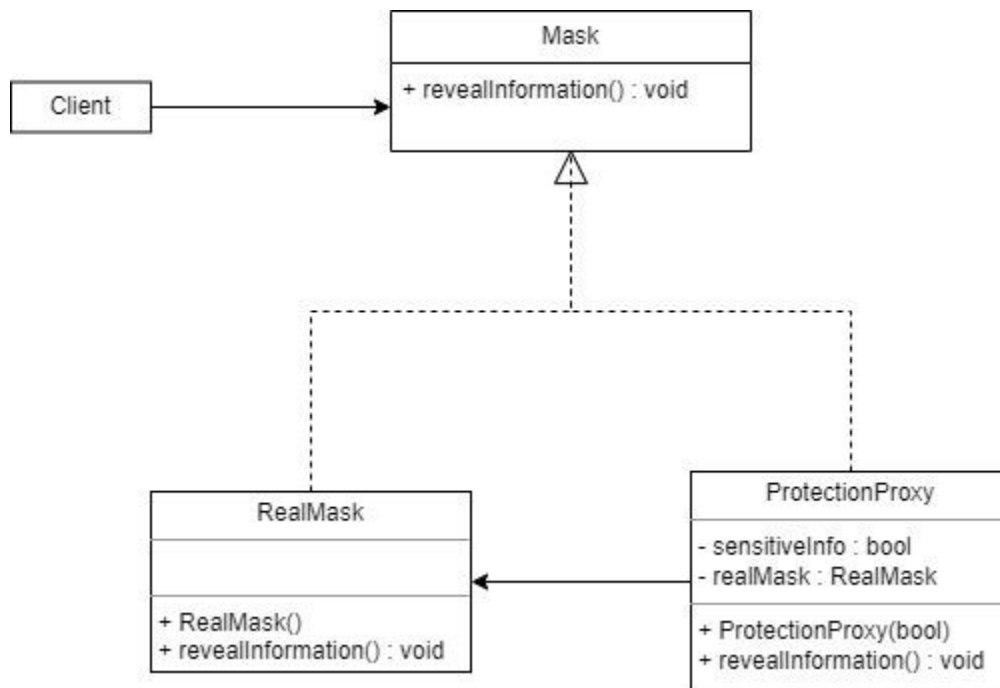
  1 usage
  show() {
    let password = "";
    let charset = this.includeUppercase + this.includeLowercase + this.includeNumbers + this.includeSpecialCharacters;

    for (let i = 0; i < this.length; i++) {
      const randomIndex = Math.floor(Math.random() * charset.length);
      password += charset.charAt(randomIndex);
    }

    return password;
  }
}

```

- Proxy Pattern:



- This pattern is a fairly simple implementation, focusing on masking and unmasking an object if the user decides to unmask or mask sensitive information. The complete construction of the class can be viewed above, and will be broken down into further detail below.

```

//Proxy
1 usage
class ProtectionProxy extends Mask {
    static sensitiveInfo;
    static realMask;

    1 usage
    constructor (isSensitiveInfoVisible) {
        super(); // Invoke the superclass constructor
        console.log("ProtectionProxy Initialized")
        this.sensitiveInfo = isSensitiveInfoVisible;
        this.realMask = new RealMask();
    }

    1 usage
    revealInformation () {
        const usernameField = document.getElementById('username');
        const passwordField = document.getElementById('password');
        const toggleSensitiveInfoButton = document.getElementById('toggleSensitiveInfo');

        if (this.sensitiveInfo) {
            this.realMask.revealInformation();
        }
        else {
            usernameField.setAttribute('type', 'password');
            passwordField.setAttribute('type', 'password');
            toggleSensitiveInfoButton.textContent = 'Unmask Sensitive Information';
        }
    }
}

const toggleButton = document.getElementById('toggleSensitiveInfo');
let isSensitiveInfoVisible = false;

toggleButton.addEventListener('click', function() {
    isSensitiveInfoVisible = !isSensitiveInfoVisible;
    console.log(isSensitiveInfoVisible.toString());
    let informationProtector = new ProtectionProxy(isSensitiveInfoVisible);
    informationProtector.revealInformation();
});

```

- ProtectionProxy protects RealMask by checking to ensure that it is receiving valid input from the logged in user in order to unmask sensitive information and display it properly. It does this by a bool variable found within the event listener, which keeps track of whether the displayed information is masked or unmasked, and waits until the user clicks the “Unmask/Mask Sensitive Information” button in order to complete the command.

```

//Real Subject
1 usage
class RealMask extends Mask {

    1 usage
    constructor () {
        super(); // Invoke the superclass constructor
        console.log("RealMask Initialized")
    }

    3 usages
    revealInformation () {
        const usernameField = document.getElementById('username');
        const passwordField = document.getElementById('password');
        const toggleSensitiveInfoButton = document.getElementById('toggleSensitiveInfo');

        usernameField.setAttribute('type', 'text');
        passwordField.setAttribute('type', 'text');
        toggleSensitiveInfoButton.textContent = 'Mask Sensitive Information';
    }
}

```

- If the information is masked and the user would like it to be unmasked, then ProtectionProxy uses RealMask in order to unmask the information without compromising the integrity of the code.

```

//Subject
6 inheritors 2 usages
class Mask {

    5+ usages
    constructor() {

    }

    6 overrides 4 usages
    revealInformation() {

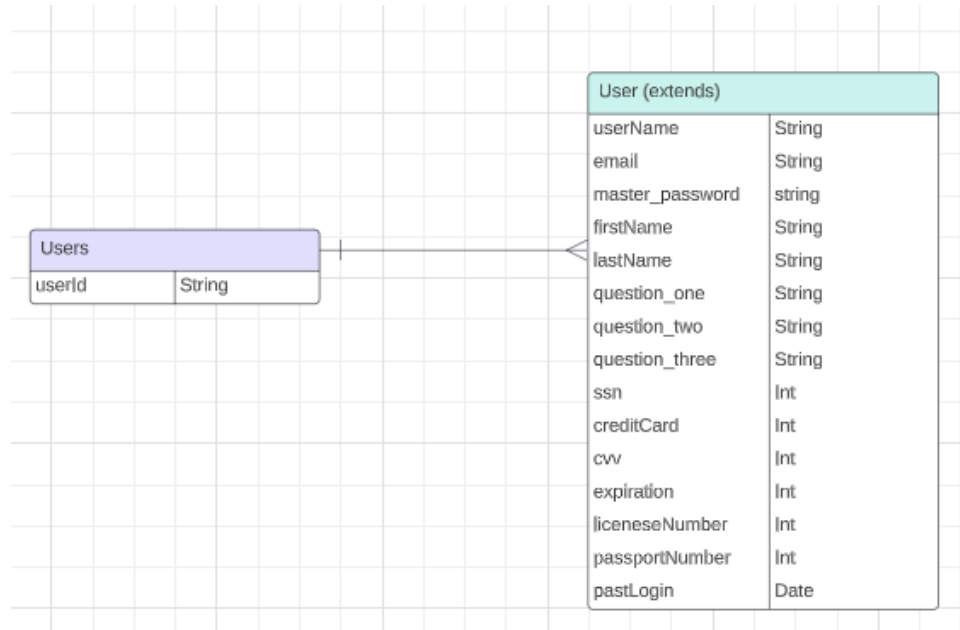
    }
}

```

- The Subject in this case is the Mask class, which defines a common interface between ProtectionProxy and RealMask in order to provide the proper flexibility and protection for when ProtectionProxy is called.

Database Schema:

To store user data, the Firebase real-time database was used. This schema represents how the database stores the information that is then accessed by the javascript to log in, register an account, and view a profile. The first object Users holds the userID(primary key), this is a unique ID created automatically when the user creates an account using this to access the user's data. The second object User extends on the first which stores all the user data including the username, email, password, security questions, and sensitive information. The database is accessed to retrieve data in all the files in the program.



```

Users
├── 2TkEfwPsxhSbw0x1hDdnc3I109G2
├── 6mZg5aKFqKb0fFt0ZUaovYt80p11
├── C0sdPTDkaLamSHdCsAmvPaNsQkU2
└── CYHEBs1Akh6ubSwQryx0BBjwUF3
    ├── creditCard: "4747185581127403"
    ├── cvv: "332"
    ├── email: "cloud@gmail.com"
    ├── expiration: "2025/02"
    ├── firstName: "John"
    ├── lastName: "Doe"
    ├── licenseNumber: "8541927"
    ├── master_password: "FDJ23kng4358ds!"
    ├── passportNumber: "187576361"
    ├── pastLogin: "2023-11-26T17:31:49.427Z"
    ├── question_one: "figures"
    ├── question_three: "artist"
    ├── question_two: "tokyojapan"
    ├── ssn: "152150615"
    └── username: "testingCloud"
  
```


User Interface:

- Index.html: user registration page. This is the first page that's displayed to the user, allowing them to create an account for MyPass. To create an account the user is required to provide a username, and email address, answer three security questions, and create a password. The username, password, and email are needed for logging into the site and the three security questions are needed for recovering the user's password. When the user interacts with the “register account” button, if all the inputs are valid the program creates an account and saves the data in the database. The generate password feature uses the builder pattern for automatically generating potential passwords for the user to use. Users are able to go to the login page directly at the bottom if they already have an existing account. It also displays information about the strength of the password if a password is inputted, informing and warning the user that the password they are using may not be strong enough, but allowing the freedom to input anything they'd like.

MyPass

Register Account

Enter Username

Enter Email Address

Security Questions

Favorite Childhood Toy or Game:

Dream Vacation Destination:

First Job or Company:

Enter Password

Generate Password

Easy to Say ☐

Easy to Read ☐

All Characters ☒

Length: 10

Register Account

Already have an account? [Log In here!](#)

yCr53)d*4C

Generate Password

Easy to Say ☐

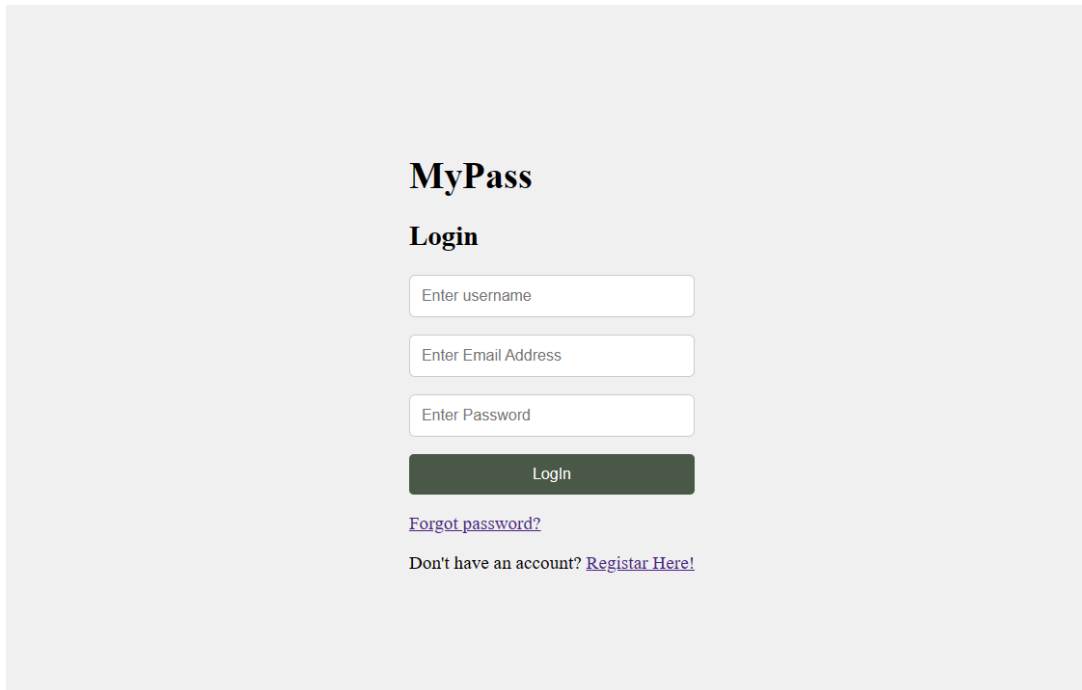
Easy to Read ☐

All Characters ☒

Length: 10

Strong

- Login.html: the login page for MyPass is simple, users are to input their username, email address, and password. Interacting with the login button will either log in the user or display an error message if the inputs are invalid. The program checks the user inputs by checking if the user account exists in the database. The singleton pattern is used for this page when the user is logging in creating one instance for the user session. If the user doesn't have an account they can use the link at the bottom to be redirected to the registration page, or if they forgot their password they can click the recover password link to be redirected to the recover password page.

The image shows a web form titled "MyPass" with a subtitle "Login". It contains three input fields: "Enter username", "Enter Email Address", and "Enter Password". Below these fields is a dark green "Login" button. At the bottom, there are two links: "Forgot password?" and "Don't have an account? Register Here!".

MyPass

Login

Enter username

Enter Email Address

Enter Password

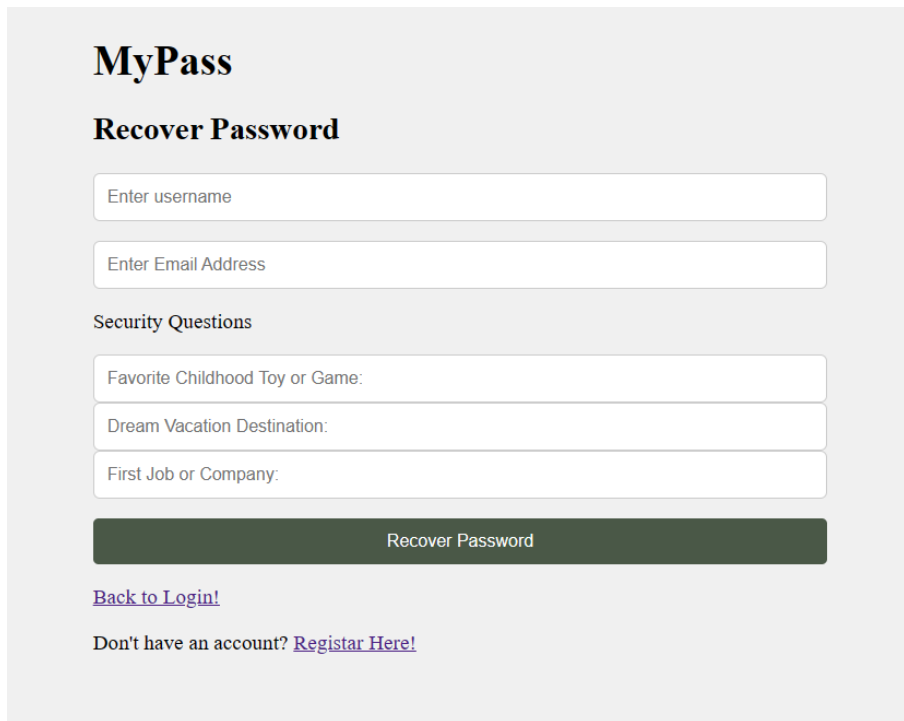
Login

[Forgot password?](#)

Don't have an account? [Register Here!](#)

- Pass.html: the recover password page follows the same style as the login page. Users who have forgotten their password have the chance to recover it by entering in their username, email and answering all three security questions correctly. If all fields are valid the password is displayed at the top in an alert for the user and they are redirected to the login page, otherwise they receive an error message. Using the chain of responsibility pattern to check if the security questions are answered correctly and to display the password. Users can go back to the login or registration page from here

using the links at the bottom.



MyPass

Recover Password

Enter username

Enter Email Address

Security Questions

Favorite Childhood Toy or Game:

Dream Vacation Destination:

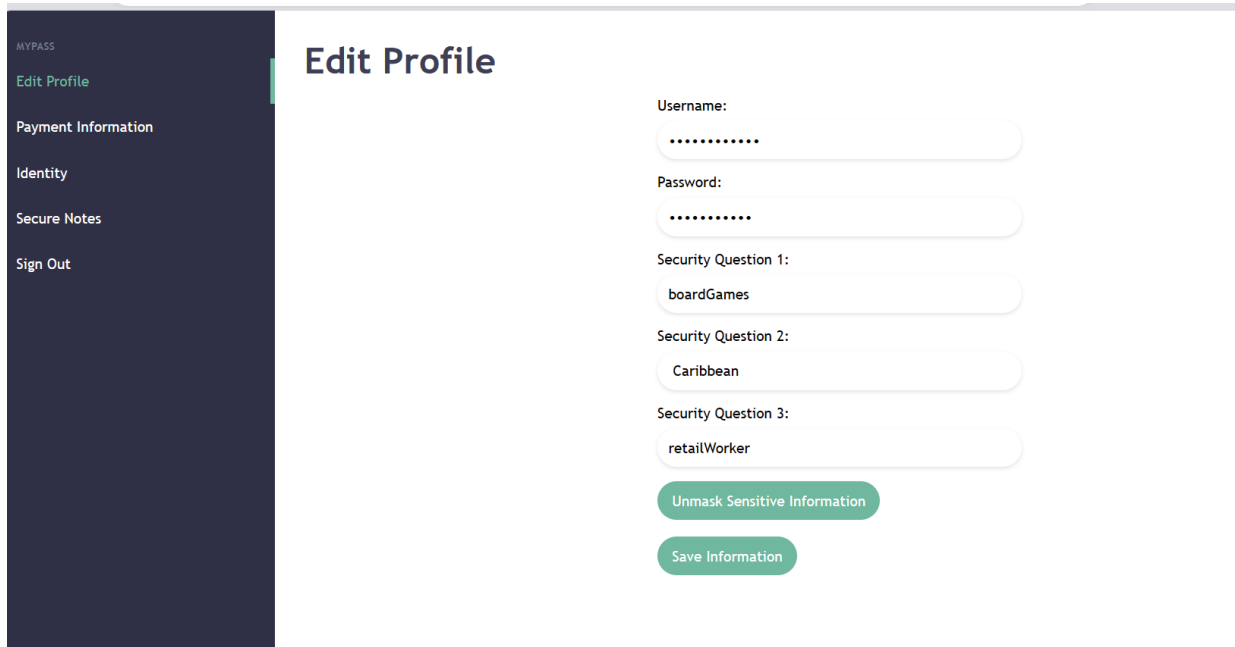
First Job or Company:

Recover Password

[Back to Login!](#)

Don't have an account? [Register Here!](#)

- Home.html: the edit profile page allows the user to view their account information such as the username, password, and security questions. Here the user is not only able to view but they can also edit the information. By saving any changes, the database is updated. The proxy pattern is used for this page to mask and unmask sensitive information such as the username and password. On the side, the user is able to access the payment information, identity, secure notes page, and sign out.



MYPASS

Edit Profile

Payment Information

Identity

Secure Notes

Sign Out

Edit Profile

Username:
.....

Password:
.....

Security Question 1:
boardGames

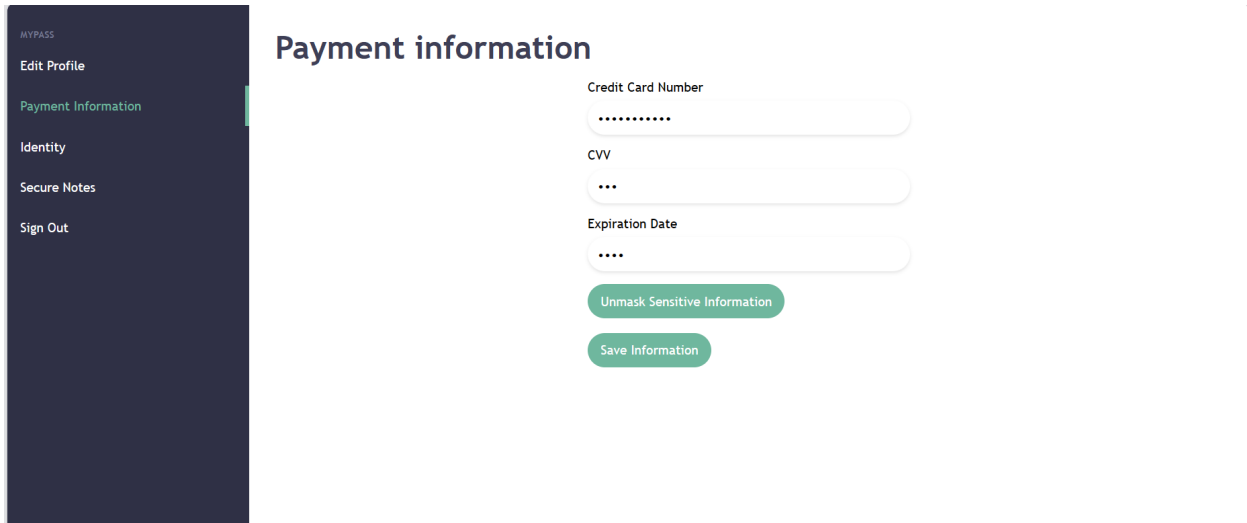
Security Question 2:
Caribbean

Security Question 3:
retailWorker

Unmask Sensitive Information

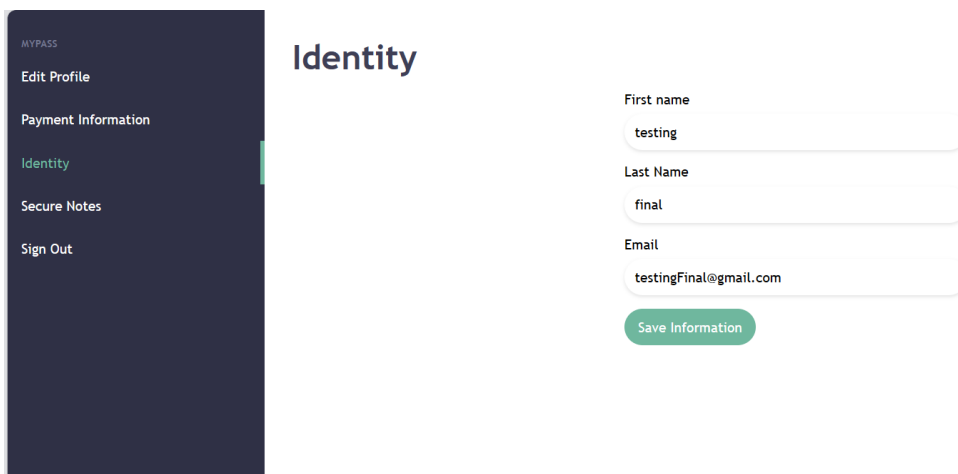
Save Information

- Payment.html: this page holds the user's credit card information, users able to view the sensitive information by interacting with the unmasking button. Like the homepage, the proxy pattern is used to mask and unmask the information, and any changes made can be saved and updated in the database.



The screenshot shows the 'Payment information' page of the MYPASS application. On the left is a dark blue sidebar with the following menu items: 'Edit Profile', 'Payment Information' (highlighted in green), 'Identity', 'Secure Notes', and 'Sign Out'. The main content area has a title 'Payment information' in bold. Below the title are three input fields: 'Credit Card Number' (containing seven dots), 'CVV' (containing three dots), and 'Expiration Date' (containing four dots). At the bottom of the form are two green buttons: 'Unmask Sensitive Information' and 'Save Information'.

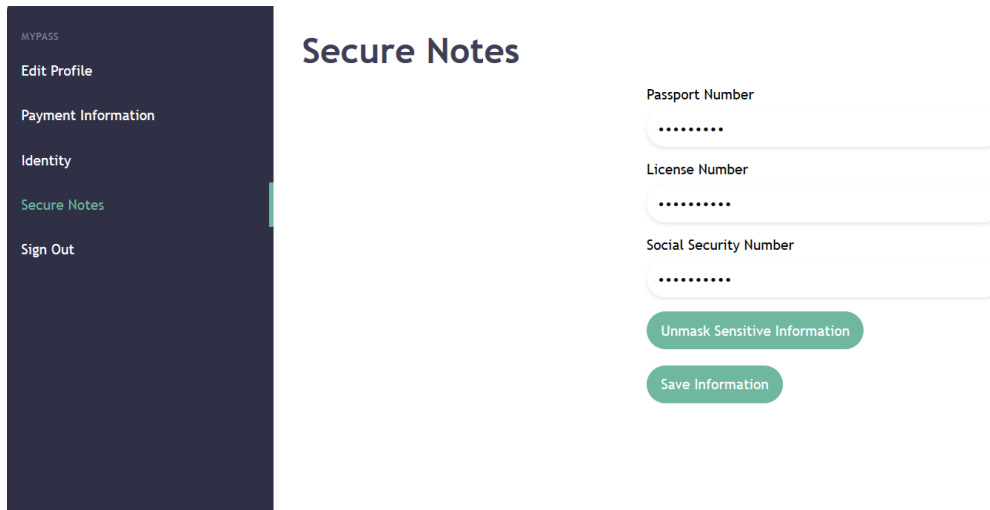
- Identity.html: this page displays the user's first name, last name, and email address. Initially, when the user first accesses this page after creating an account the fields first name and last name are empty thus the user needs to enter that information in and save. After saving the information, the program updates the database and will display the information.



The screenshot shows the 'Identity' page of the MYPASS application. On the left is a dark blue sidebar with the following menu items: 'Edit Profile', 'Payment Information', 'Identity' (highlighted in green), 'Secure Notes', and 'Sign Out'. The main content area has a title 'Identity' in bold. Below the title are three input fields: 'First name' (containing the text 'testing'), 'Last Name' (containing the text 'final'), and 'Email' (containing the text 'testingFinal@gmail.com'). At the bottom of the form is a green button labeled 'Save Information'.

- Secure.html: this page holds the user's passport, license, and social security number, all these fields are initially empty and the user is allowed to enter the information and save it

to the database. The information on the page is masked using the proxy pattern but could be unmasked using the “unmask” button. Altering any of the fields and saving will update the database.



The screenshot displays a web application interface. On the left is a dark blue sidebar with the text 'MYPASS' at the top. Below it are menu items: 'Edit Profile', 'Payment Information', 'Identity', 'Secure Notes' (highlighted in green), and 'Sign Out'. The main content area has a white background with the heading 'Secure Notes' in a bold, dark blue font. Below the heading are three input fields, each with a label and a masked value (seven dots): 'Passport Number', 'License Number', and 'Social Security Number'. At the bottom of the form are two green buttons: 'Unmask Sensitive Information' and 'Save Information'.

References:

“Manage Users in Firebase.” Firebase,
<https://firebase.google.com/docs/auth/web/manage-users>. Accessed 26 November 2023.