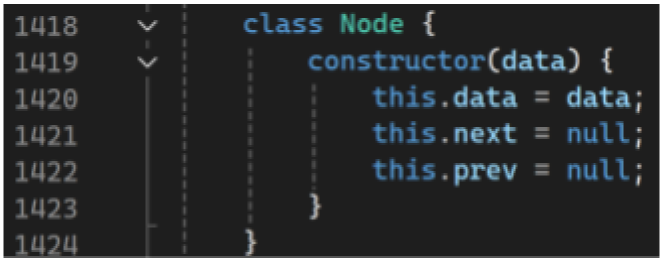
Criterion C: Development

Techniques Used

- Linked lists data structure
- Data transfer object
- Parent-child relationships
- Hashed password
- Session data
- Jquery
- Dapper
- Razor pages

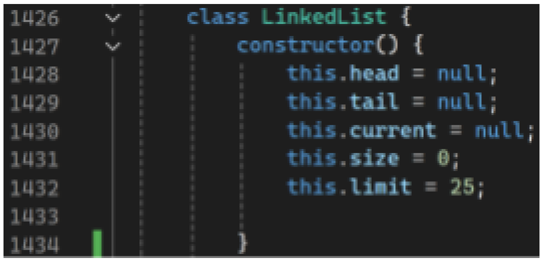
Linked lists data structure

One of the functions of the Excel file, which my client previously used to keep track of his rewards, was the undo feature. Thus, I created a linked list data structure (Figures 1-4) and implemented it to create multiple history stacks—one for each table (Figure 5). This allows the client to undo each type of action and undo actions before the most recent action. All of this gives the client more flexibility and convenience, helping the user save time.



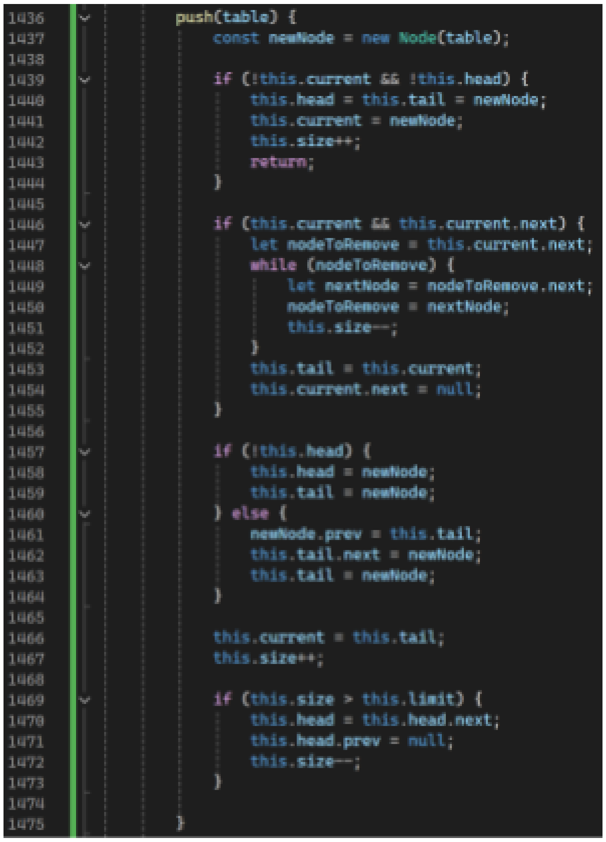
```
class Node {
    constructor(data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
```

Figure 1: Node constructor



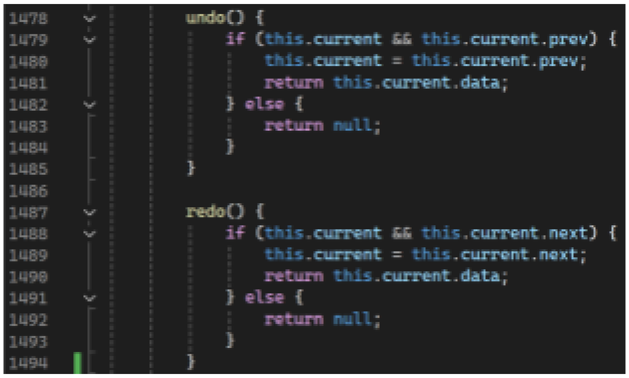
```
class LinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.current = null;
        this.size = 0;
        this.limit = 25;

    }
```

Figure 2: LinkedList constructor



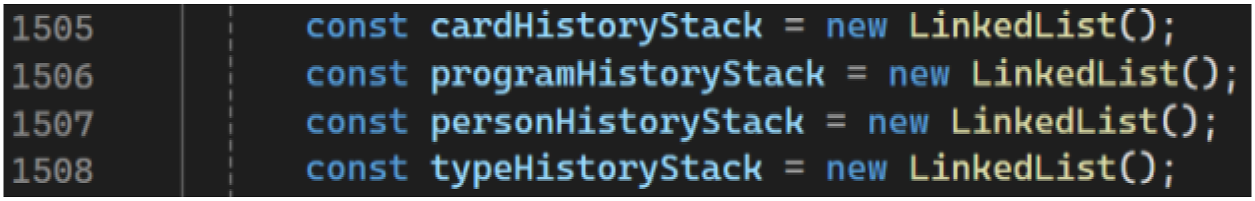
```
    push(table) {
        const newNode = new Node(table);

        if (!this.current && !this.head) {
            this.head = this.tail = newNode;
            this.current = newNode;
            this.size++;
            return;
        }

        if (this.current && this.current.next) {
            let nodeToRemove = this.current.next;
            while (nodeToRemove) {
                let nextNode = nodeToRemove.next;
                nodeToRemove = nextNode;
                this.size--;
            }
            this.tail = this.current;
            this.current.next = null;
        }

        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            newNode.prev = this.tail;
            this.tail.next = newNode;
            this.tail = newNode;
        }

        this.current = this.tail;
        this.size++;

        if (this.size > this.limit) {
            this.head = this.head.next;
            this.head.prev = null;
            this.size--;
        }

    }
```

Figure 3: LinkedList push function



```
    undo() {
        if (this.current && this.current.prev) {
            this.current = this.current.prev;
            return this.current.data;
        } else {
            return null;
        }
    }

    redo() {
        if (this.current && this.current.next) {
            this.current = this.current.next;
            return this.current.data;
        } else {
            return null;
        }
    }
```

Figure 4: LinkedList undo function and redo function

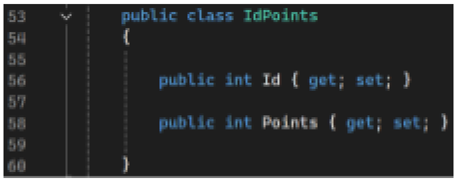


```
    const cardHistoryStack = new LinkedList();
    const programHistoryStack = new LinkedList();
    const personHistoryStack = new LinkedList();
    const typeHistoryStack = new LinkedList();
```

Figure 5: History stacks

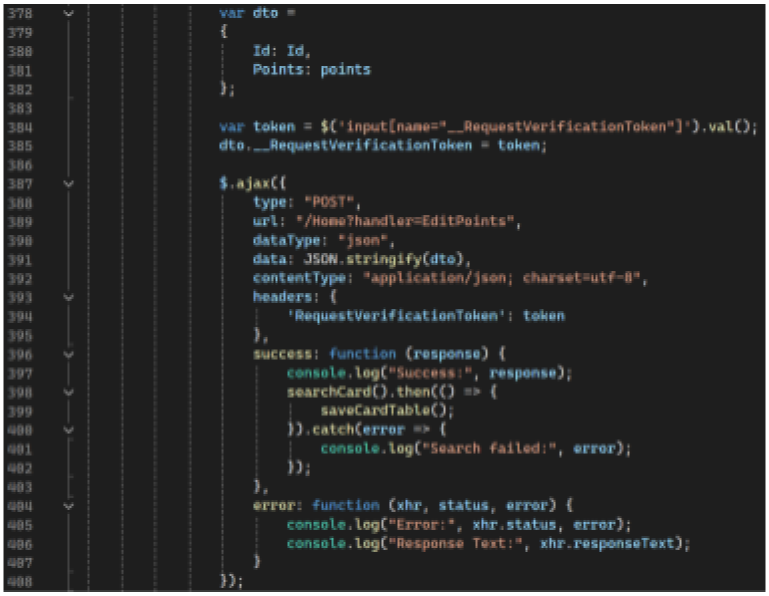
Data transfer object

In order to help with making the points more convenient to edit, a data transfer class (Figure 6) and object (Figure 7) was created so that the points attribute on a card can be edited from the main table's edit points text field. Passing in the data this way allows the user to get the Request Verification Token needed to make the ajax call. If a Request Verification Token is not passed through, the ajax call does not go through.



```
public class IdPoints
{

    public int Id { get; set; }

    public int Points { get; set; }

}
```

Figure 6: DTO class

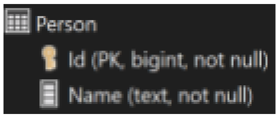


```
var dto =
{
    Id: Id,
    Points: points
};

var token = $('input[name="__RequestVerificationToken"]').val();
dto.__RequestVerificationToken = token;

$.ajax({
    type: "POST",
    url: "/Home?handler=EditPoints",
    dataType: "json",
    data: JSON.stringify(dto),
    contentType: "application/json; charset=utf-8",
    headers: {
        'RequestVerificationToken': token
    },
    success: function (response) {
        console.log("Success:", response);
        searchCard().then(() => {
            saveCardTable();
        }).catch(error => {
            console.log("Search failed:", error);
        });
    },
    error: function (xhr, status, error) {
        console.log("Error:", xhr.status, error);
        console.log("Response Text:", xhr.responseText);
    }
});
```

Figure 7: Passing a IdPoints DTO

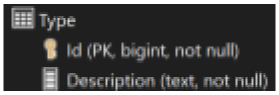
Parent-child relationships

To better organize the nature of the data after some consultations with the client and my advisor (Appendix B), the card table was split up into four other tables with CRUD functionality (Figures 8-11). This allowed for less repetitiveness, as instead of re-inputting a program’s data for each card, a program’s data can be assigned to many cards while only being entered once (Figure 12).



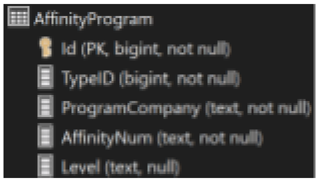
Person
- Id (PK, bigint, not null)
- Name (text, not null)

Figure 8: Person Model



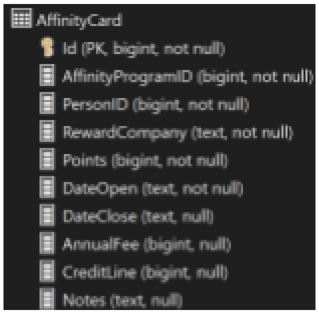
Type
- Id (PK, bigint, not null)
- Description (text, not null)

Figure 9: Type Model



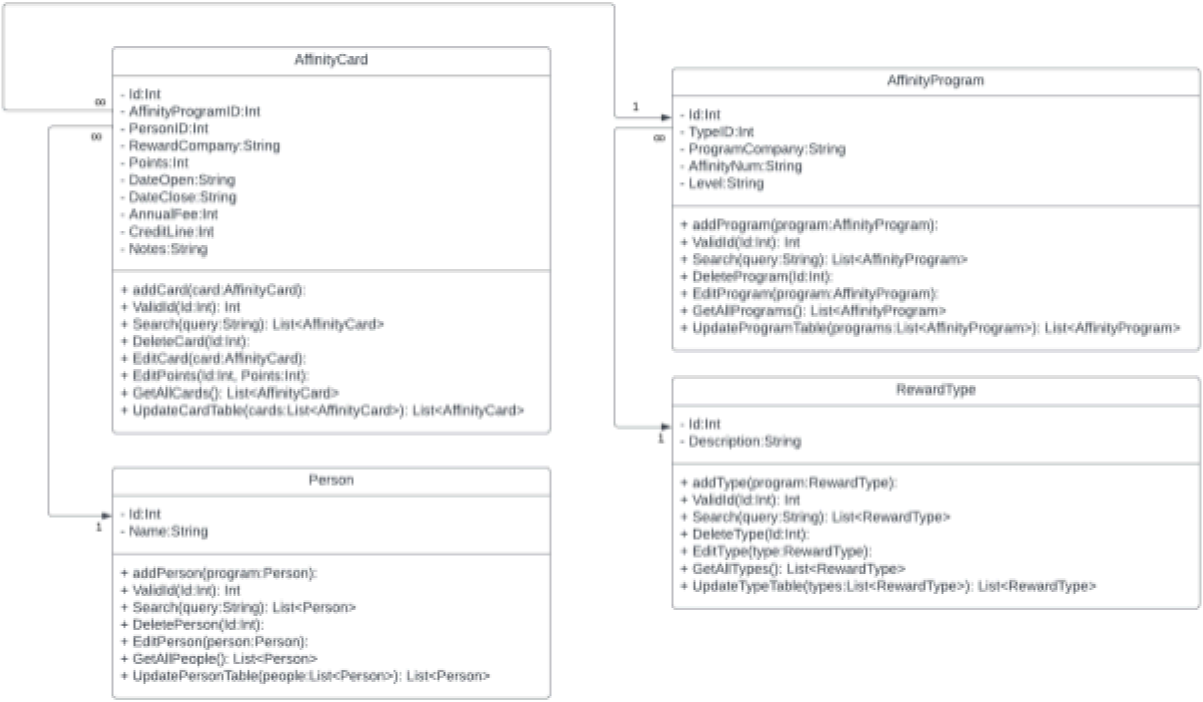
AffinityProgram
- Id (PK, bigint, not null)
- TypeID (bigint, not null)
- ProgramCompany (text, not null)
- AffinityNum (text, not null)
- Level (text, null)

Figure 10: AffinityProgram Model



AffinityCard
- Id (PK, bigint, not null)
- AffinityProgramID (bigint, not null)
- PersonID (bigint, not null)
- RewardCompany (text, not null)
- Points (bigint, not null)
- DateOpen (text, not null)
- DateClose (text, null)
- AnnualFee (bigint, null)
- CreditLine (bigint, null)
- Notes (text, null)

Figure 11: AffinityCard Model



AffinityCard

```
- Id:Int
- AffinityProgramID:Int
- PersonID:Int
- RewardCompany:String
- Points:Int
- DateOpen:String
- DateClose:String
- AnnualFee:Int
- CreditLine:Int
- Notes:String

+ addCard(card:AffinityCard):
+ ValidId(Id:Int): Int
+ Search(query:String): List<AffinityCard>
+ DeleteCard(Id:Int):
+ EditCard(card:AffinityCard):
+ EditPoints(Id:Int, Points:Int):
+ GetAllCards(): List<AffinityCard>
+ UpdateCardTable(cards:List<AffinityCard>): List<AffinityCard>
```

Person

```
- Id:Int
- Name:String

+ addPerson(program:Person):
+ ValidId(Id:Int): Int
+ Search(query:String): List<Person>
+ DeletePerson(Id:Int):
+ EditPerson(person:Person):
+ GetAllPeople(): List<Person>
+ UpdatePersonTable(people:List<Person>): List<Person>
```

AffinityProgram

```
- Id:Int
- TypeID:Int
- ProgramCompany:String
- AffinityNum:String
- Level:String

+ addProgram(program:AffinityProgram):
+ ValidId(Id:Int): Int
+ Search(query:String): List<AffinityProgram>
+ DeleteProgram(Id:Int):
+ EditProgram(program:AffinityProgram):
+ GetAllPrograms(): List<AffinityProgram>
+ UpdateProgramTable(programs:List<AffinityProgram>): List<AffinityProgram>
```

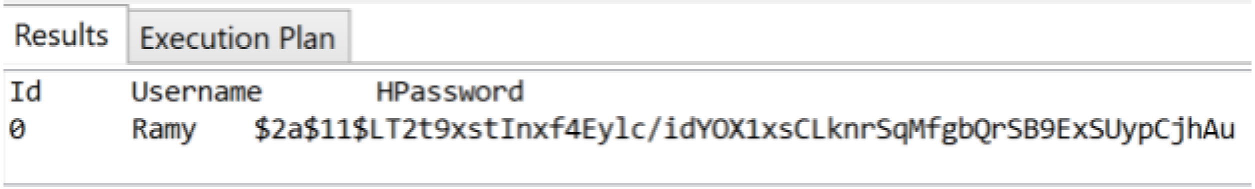
RewardType

```
- Id:Int
- Description:String

+ addType(program:RewardType):
+ ValidId(Id:Int): Int
+ Search(query:String): List<RewardType>
+ DeleteType(Id:Int):
+ EditType(type:RewardType):
+ GetAllTypes(): List<RewardType>
+ UpdateTypeTable(types:List<RewardType>): List<RewardType>
```

Figure 12: Models UML Diagram

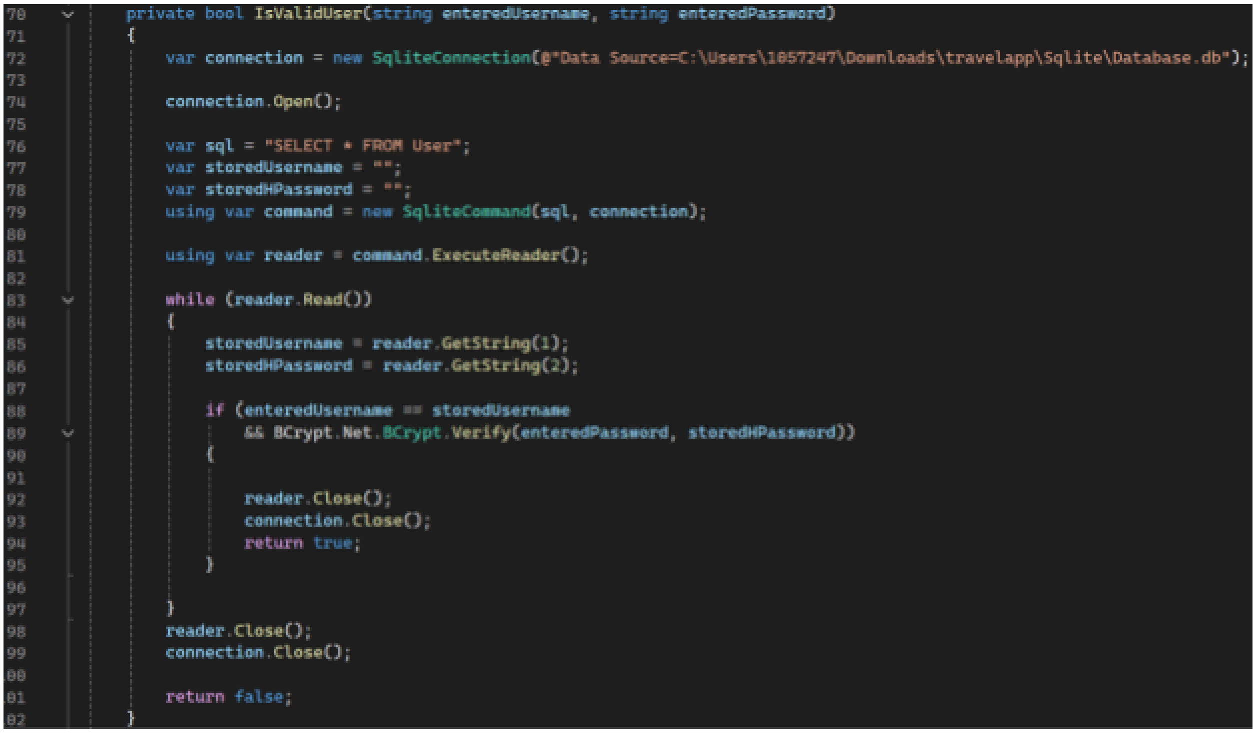
Hashed password

Hashing the password strengthens the security of the web application, which fulfills the success criteria of protecting the web application from users other than the client. Below is an image of the login credentials stored inside the database (Figure 13) and the function to check whether or not the user has inputted the correct login credentials (Figure 14). The hashed out password is not what the user enters into the password textfield in the login page, but an encrypted version of it. The commented out code in Figure 16 is how I encrypted the data. Thus, even if a user that is not my client managed to peek into the database, they would not be able to login by looking at the hashed password.



```
Id      Username        HPassword
0       Ramy    $2a$11$LT2t9xstInxf4Eylc/idYOX1xsCLknrSqMfgbQrSB9ExSUypCjhAu
```

Figure 13: Login credentials query

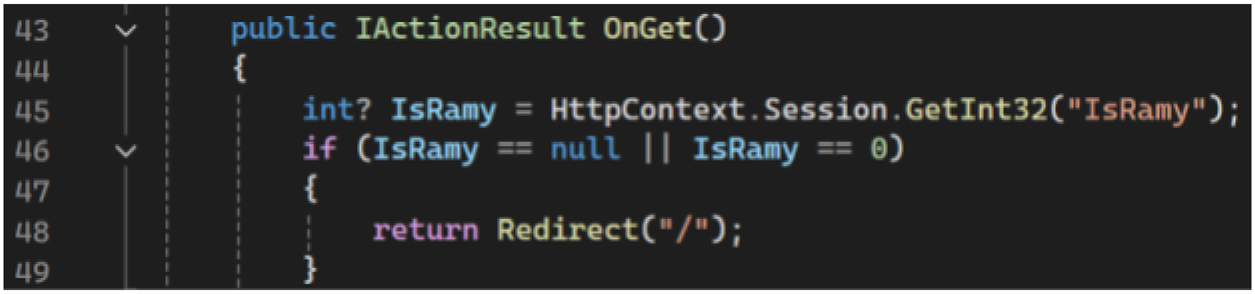


```
private bool IsValidUser(string enteredUsername, string enteredPassword)
{
    var connection = new SqliteConnection(@"Data Source=C:\Users\1057247\Downloads\travelapp\Sqlite\Database.db");

    connection.Open();

    var sql = "SELECT * FROM User";
    var storedUsername = "";
    var storedHPassword = "";
    using var command = new SqliteCommand(sql, connection);

    using var reader = command.ExecuteReader();

    while (reader.Read())
    {
        storedUsername = reader.GetString(1);
        storedHPassword = reader.GetString(2);

        if (enteredUsername == storedUsername
            && BCrypt.Net.BCrypt.Verify(enteredPassword, storedHPassword))
        {

            reader.Close();
            connection.Close();
            return true;
        }

    }
    reader.Close();
    connection.Close();

    return false;
}
```

Figure 14: Login page IsValidUser function

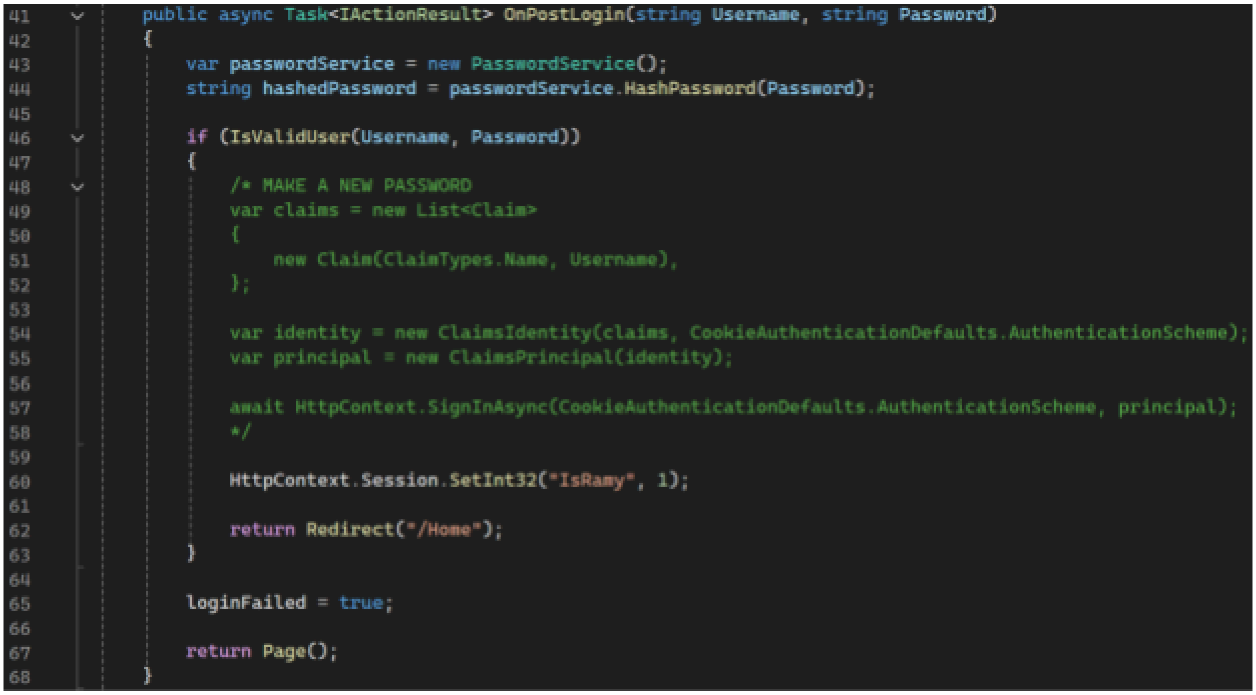
Session data

Storing data in session variables has two functions. The first is security. If the user enters the page without the session variable "IsRamy" being 1, which represents true, it sends them back to the login page (Figure 15). Essentially, it prevents users from typing "/Home" into the URL and bypassing inputting the username and password on the login page. If the user successfully enters the correct login credentials, it sets "IsRamy" to 1 and redirects the user to the home page (Figure 16). The second use is storing and retrieving history data (Figures 17-18). Because the history data is only meant to be temporary, session variables are perfect, as they clear after the user closes the browser. A javascript variable would not be sufficient enough to store stack data, as every time the user refreshes, the history data is cleared. Because it cannot tell the difference between when the user first enters the web application and when the user has just refreshed the page, when the variable is first declared it also clears all data from that variable upon refreshing. Thus, a session variable is great for the task of storing history data.



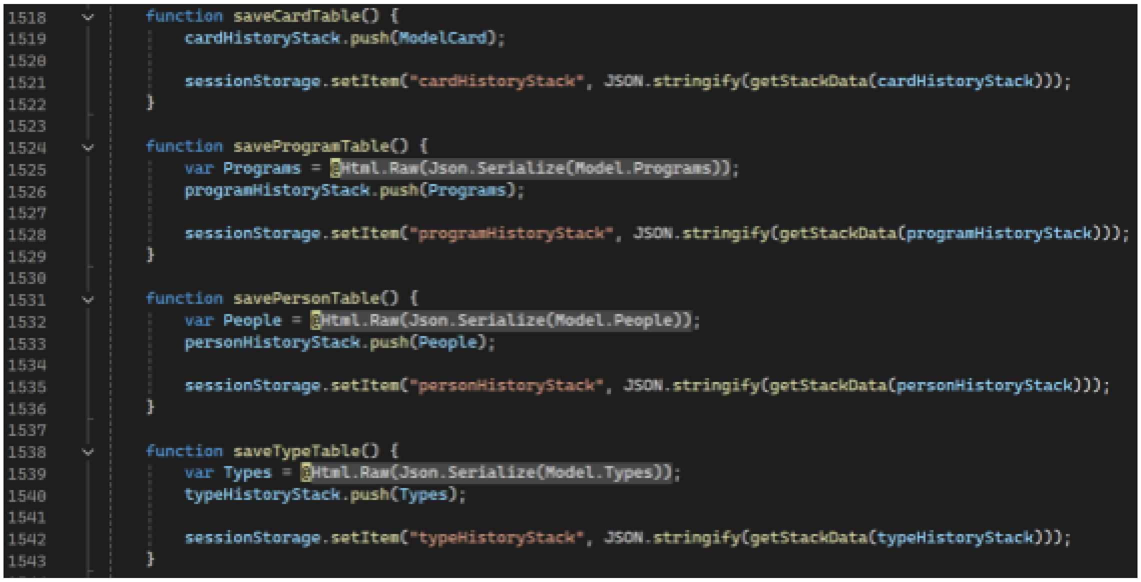
```
public IActionResult OnGet()
{
    int? IsRamy = HttpContext.Session.GetInt32("IsRamy");
    if (IsRamy == null || IsRamy == 0)
    {
        return Redirect("/");
    }
```

Figure 15: Home page OnGet function



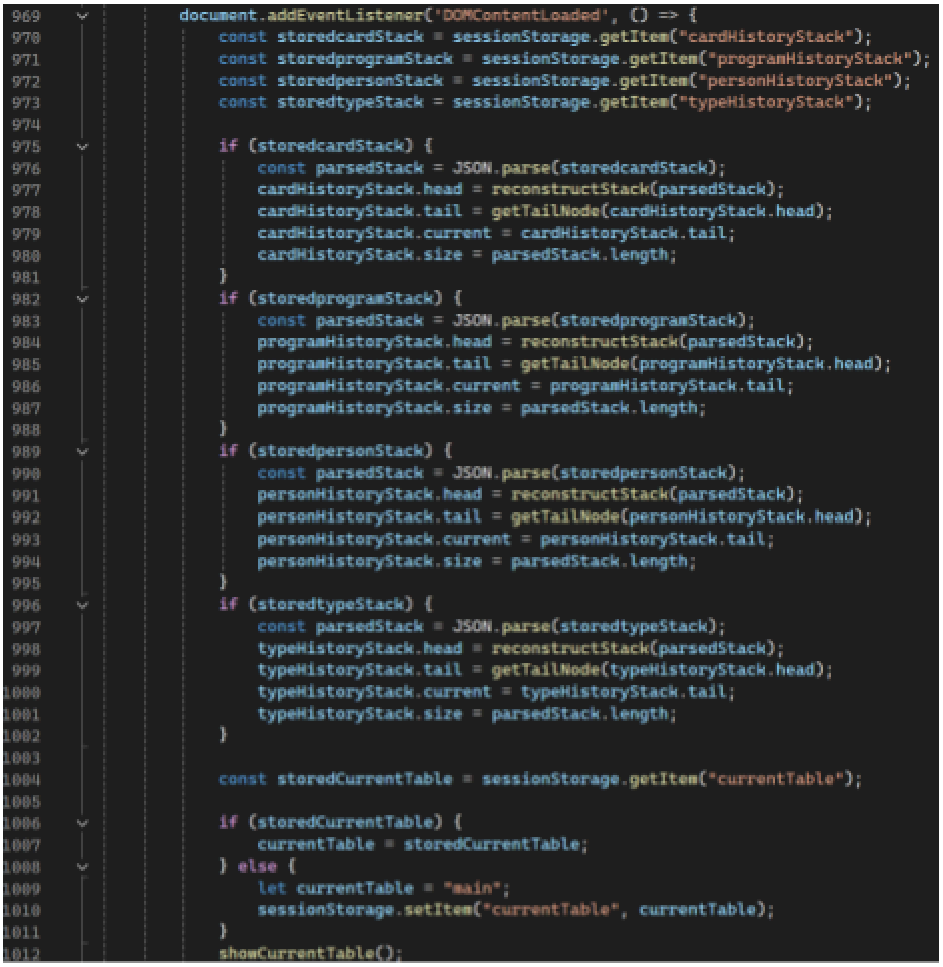
```
public async Task<IActionResult> OnPostLogin(string Username, string Password)
{
    var passwordService = new PasswordService();
    string hashedPassword = passwordService.HashPassword(Password);

    if (IsValidUser(Username, Password))
    {
        /* MAKE A NEW PASSWORD
        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.Name, Username),
        };

        var identity = new ClaimsIdentity(claims, CookieAuthenticationDefaults.AuthenticationScheme);
        var principal = new ClaimsPrincipal(identity);

        await HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, principal);
        */

        HttpContext.Session.SetInt32("IsRamy", 1);

        return Redirect("/Home");
    }

    loginFailed = true;

    return Page();
}
```

Figure 16: Login page OnPostLogin function



```
function saveCardTable() {
    cardHistoryStack.push(ModelCard);

    sessionStorage.setItem("cardHistoryStack", JSON.stringify(getStackData(cardHistoryStack)));
}

function saveProgramTable() {
    var Programs = @Html.Raw(Json.Serialize(Model.Programs));
    programHistoryStack.push(Programs);

    sessionStorage.setItem("programHistoryStack", JSON.stringify(getStackData(programHistoryStack)));
}

function savePersonTable() {
    var People = @Html.Raw(Json.Serialize(Model.People));
    personHistoryStack.push(People);

    sessionStorage.setItem("personHistoryStack", JSON.stringify(getStackData(personHistoryStack)));
}

function saveTypeTable() {
    var Types = @Html.Raw(Json.Serialize(Model.Types));
    typeHistoryStack.push(Types);

    sessionStorage.setItem("typeHistoryStack", JSON.stringify(getStackData(typeHistoryStack)));
}
```

Figure 17: Updating history stacks

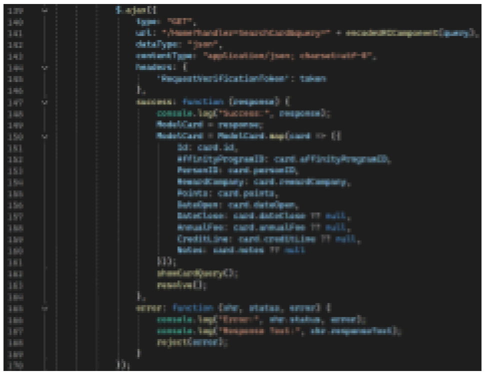


```
document.addEventListener('DOMContentLoaded', () => {
    const storedcardStack = sessionStorage.getItem("cardHistoryStack");
    const storedprogramStack = sessionStorage.getItem("programHistoryStack");
    const storedpersonStack = sessionStorage.getItem("personHistoryStack");
    const storedtypeStack = sessionStorage.getItem("typeHistoryStack");

    if (storedcardStack) {
        const parsedStack = JSON.parse(storedcardStack);
        cardHistoryStack.head = reconstructStack(parsedStack);
        cardHistoryStack.tail = getTailNode(cardHistoryStack.head);
        cardHistoryStack.current = cardHistoryStack.tail;
        cardHistoryStack.size = parsedStack.length;
    }
    if (storedprogramStack) {
        const parsedStack = JSON.parse(storedprogramStack);
        programHistoryStack.head = reconstructStack(parsedStack);
        programHistoryStack.tail = getTailNode(programHistoryStack.head);
        programHistoryStack.current = programHistoryStack.tail;
        programHistoryStack.size = parsedStack.length;
    }
    if (storedpersonStack) {
        const parsedStack = JSON.parse(storedpersonStack);
        personHistoryStack.head = reconstructStack(parsedStack);
        personHistoryStack.tail = getTailNode(personHistoryStack.head);
        personHistoryStack.current = personHistoryStack.tail;
        personHistoryStack.size = parsedStack.length;
    }
    if (storedtypeStack) {
        const parsedStack = JSON.parse(storedtypeStack);
        typeHistoryStack.head = reconstructStack(parsedStack);
        typeHistoryStack.tail = getTailNode(typeHistoryStack.head);
        typeHistoryStack.current = typeHistoryStack.tail;
        typeHistoryStack.size = parsedStack.length;
    }

    const storedCurrentTable = sessionStorage.getItem("currentTable");

    if (storedCurrentTable) {
        currentTable = storedCurrentTable;
    } else {
        let currentTable = "main";
        sessionStorage.setItem("currentTable", currentTable);
    }
    showCurrentTable();
```

Figure 18: Retrieving history stacks

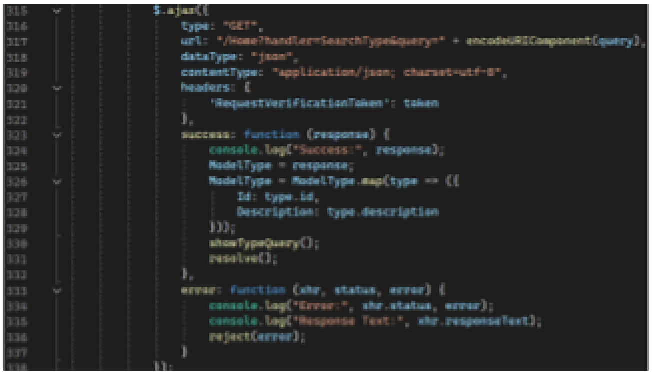
Jquery

The Jquery library allows for ajax calls, which better organizes the code and can retrieve or assign data without the page reloading, giving my client a more seamless experience. Figures 7 and 19-30 show the edit points call for the card table and the search, update, and delete calls for the card, program, person, and type tables. The POST functions (update, delete and edit points) require a Request Verification Token and will not run if it does not have the token, which adds an extra layer of security to the web application as it prevents CSRF (Cross-Site Request Forgery) attacks.



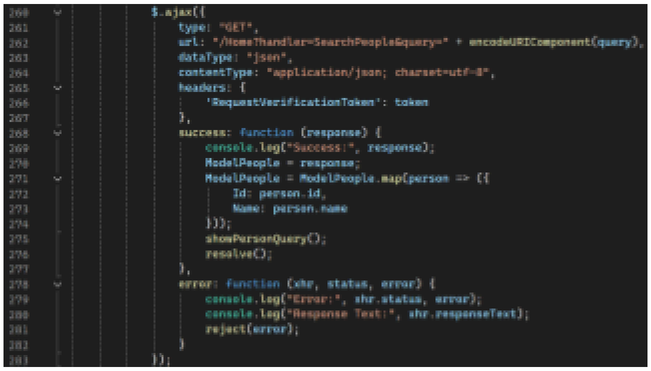
```
$.ajax({
    type: "GET",
    url: "/Home?handler=SearchCard&query=" + encodeURIComponent(query),
    dataType: "json",
    contentType: "application/json; charset=utf-8",
    headers: {
        'RequestVerificationToken': token
    },
    success: function (response) {
        console.log("Success:", response);
        ModelCard = response;
        ModelCard = ModelCard.map(card => ({
            Id: card.id,
            AffinityProgramID: card.affinityProgramID,
            PersonID: card.personID,
            RewardCompany: card.rewardCompany,
            Points: card.points,
            DateOpen: card.dateOpen,
            DateClose: card.dateClose ?? null,
            AnnualFee: card.annualFee ?? null,
            CreditLine: card.creditLine ?? null,
            Notes: card.notes ?? null
        }));
        showCardQuery();
        resolve();
    },
    error: function (xhr, status, error) {
        console.log("Error:", xhr.status, error);
        console.log("Response Text:", xhr.responseText);
        reject(error);
    }
});
```

Figure 19: SearchCard ajax call



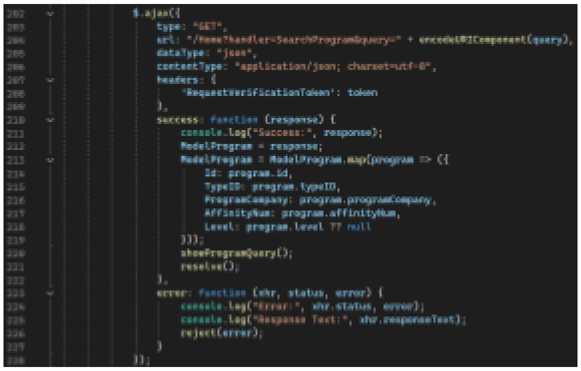
```
$.ajax({
    type: "GET",
    url: "/Home?handler=SearchType&query=" + encodeURIComponent(query),
    dataType: "json",
    contentType: "application/json; charset=utf-8",
    headers: {
        'RequestVerificationToken': token
    },
    success: function (response) {
        console.log("Success:", response);
        ModelType = response;
        ModelType = ModelType.map(type => ({
            Id: type.id,
            Description: type.description
        }));
        showTypeQuery();
        resolve();
    },
    error: function (xhr, status, error) {
        console.log("Error:", xhr.status, error);
        console.log("Response Text:", xhr.responseText);
        reject(error);
    }
});
```

Figure 20: SearchProgram ajax call



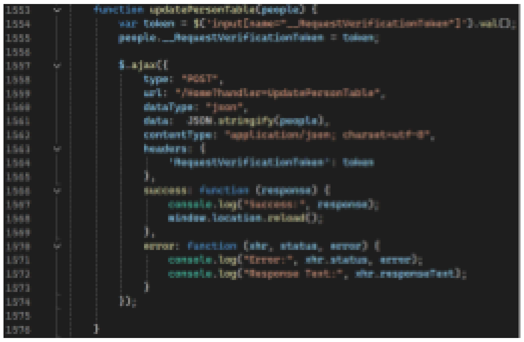
```
$.ajax({
    type: "GET",
    url: "/Home?handler=SearchPeople&query=" + encodeURIComponent(query),
    dataType: "json",
    contentType: "application/json; charset=utf-8",
    headers: {
        'RequestVerificationToken': token
    },
    success: function (response) {
        console.log("Success:", response);
        ModelPeople = response;
        ModelPeople = ModelPeople.map(person => ({
            Id: person.id,
            Name: person.name
        }));
        showPersonQuery();
        resolve();
    },
    error: function (xhr, status, error) {
        console.log("Error:", xhr.status, error);
        console.log("Response Text:", xhr.responseText);
        reject(error);
    }
});
```

Figure 21: SearchPeople ajax call



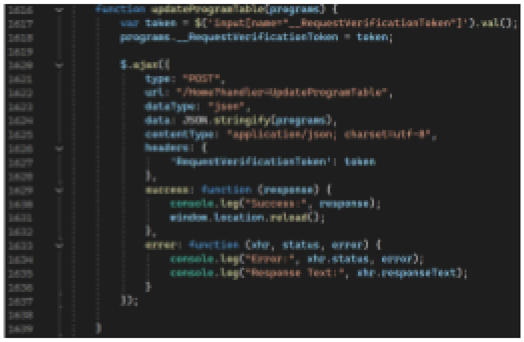
```
$.ajax({
    type: "GET",
    url: "/Home?handler=SearchProgram&query=" + encodeURIComponent(query),
    dataType: "json",
    contentType: "application/json; charset=utf-8",
    headers: {
        'RequestVerificationToken': token
    },
    success: function (response) {
        console.log("Success:", response);
        ModelProgram = response;
        ModelProgram = ModelProgram.map(program => ({
            Id: program.id,
            TypeID: program.typeID,
            ProgramCompany: program.programCompany,
            AffinityNum: program.affinityNum,
            Level: program.level ?? null
        }));
        showProgramQuery();
        resolve();
    },
    error: function (xhr, status, error) {
        console.log("Error:", xhr.status, error);
        console.log("Response Text:", xhr.responseText);
        reject(error);
    }
});
```

Figure 22: SearchType ajax call



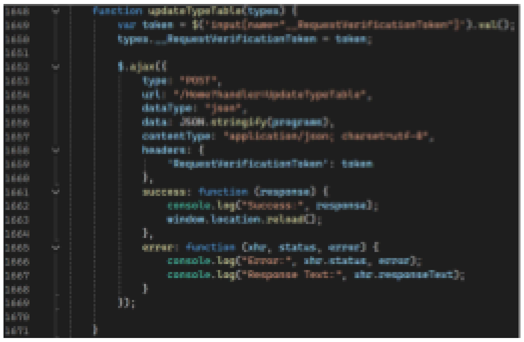
```
function updatePersonTable(people) {
    var token = $('input[name="__RequestVerificationToken"]').val();
    people.__RequestVerificationToken = token;

    $.ajax({
        type: "POST",
        url: "/Home?handler=UpdatePersonTable",
        dataType: "json",
        data:  JSON.stringify(people),
        contentType: "application/json; charset=utf-8",
        headers: {
            'RequestVerificationToken': token
        },
        success: function (response) {
            console.log("Success:", response);
            window.location.reload();
        },
        error: function (xhr, status, error) {
            console.log("Error:", xhr.status, error);
            console.log("Response Text:", xhr.responseText);
        }
    });

}
```

Figure 23: updatePersonTable ajax call



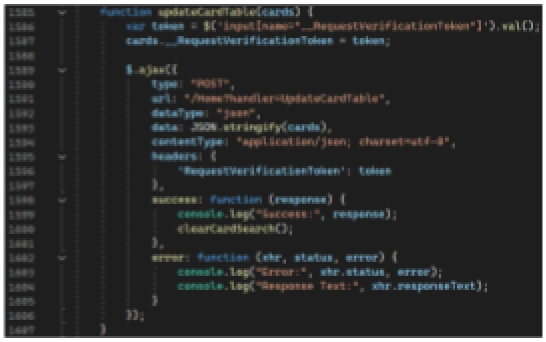
```
function updateProgramTable(programs) {
    var token = $('input[name="__RequestVerificationToken"]').val();
    programs.__RequestVerificationToken = token;

    $.ajax({
        type: "POST",
        url: "/Home?handler=UpdateProgramTable",
        dataType: "json",
        data: JSON.stringify(programs),
        contentType: "application/json; charset=utf-8",
        headers: {
            'RequestVerificationToken': token
        },
        success: function (response) {
            console.log("Success:", response);
            window.location.reload();
        },
        error: function (xhr, status, error) {
            console.log("Error:", xhr.status, error);
            console.log("Response Text:", xhr.responseText);
        }
    });

}
```

Figure 24: updateProgramTable ajax call



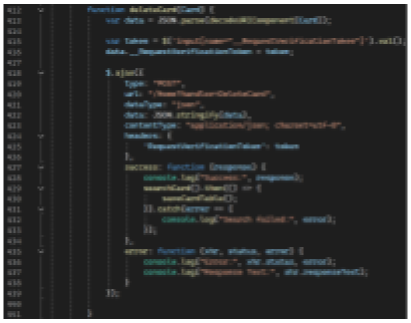
```
function updateTypeTable(types) {
    var token = $('input[name="__RequestVerificationToken"]').val();
    types.__RequestVerificationToken = token;

    $.ajax({
        type: "POST",
        url: "/Home?handler=UpdateTypeTable",
        dataType: "json",
        data: JSON.stringify(programs),
        contentType: "application/json; charset=utf-8",
        headers: {
            'RequestVerificationToken': token
        },
        success: function (response) {
            console.log("Success:", response);
            window.location.reload();
        },
        error: function (xhr, status, error) {
            console.log("Error:", xhr.status, error);
            console.log("Response Text:", xhr.responseText);
        }
    });

}
```

Figure 25: updateTypeTable ajax call



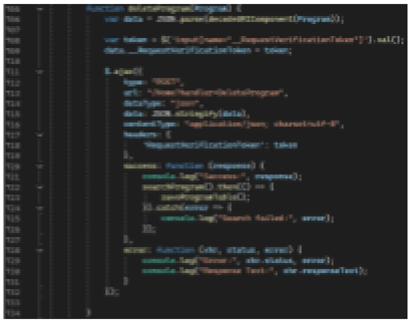
```
function updateCardTable(cards) {
    var token = $('input[name="__RequestVerificationToken"]').val();
    cards.__RequestVerificationToken = token;

    $.ajax({
        type: "POST",
        url: "/Home?handler=UpdateCardTable",
        dataType: "json",
        data: JSON.stringify(cards),
        contentType: "application/json; charset=utf-8",
        headers: {
            'RequestVerificationToken': token
        },
        success: function (response) {
            console.log("Success:", response);
            clearCardSearch();
        },
        error: function (xhr, status, error) {
            console.log("Error:", xhr.status, error);
            console.log("Response Text:", xhr.responseText);
        }
    });
}
```

Figure 26: updateCardTable ajax call



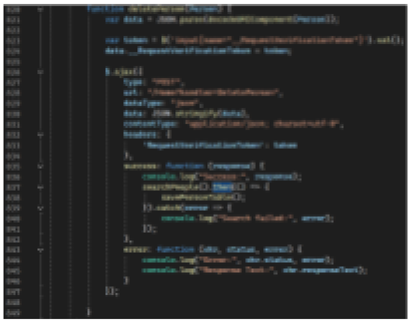
```
function deleteCard(Card) {
    var data = JSON.parse(decodeURIComponent(Card));

    var token = $('input[name="__RequestVerificationToken"]').val();
    data.__RequestVerificationToken = token;

    $.ajax({
        type: "POST",
        url: "/Home?handler=DeleteCard",
        dataType: "json",
        data: JSON.stringify(data),
        contentType: "application/json; charset=utf-8",
        headers: {
            'RequestVerificationToken': token
        },
        success: function (response) {
            console.log("Success:", response);
            searchCard().then(() => {
                saveCardTable();
            }).catch(error => {
                console.log("Search failed:", error);
            });
        },
        error: function (xhr, status, error) {
            console.log("Error:", xhr.status, error);
            console.log("Response Text:", xhr.responseText);
        }
    });

}
```

Figure 27: DeleteCard ajax call



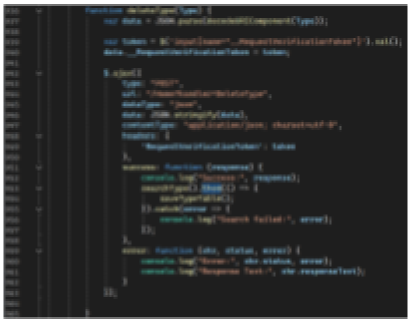
```
function deleteProgram(Program) {
    var data = JSON.parse(decodeURIComponent(Program));

    var token = $('input[name="__RequestVerificationToken"]').val();
    data.__RequestVerificationToken = token;

    $.ajax({
        type: "POST",
        url: "/Home?handler=DeleteProgram",
        dataType: "json",
        data: JSON.stringify(data),
        contentType: "application/json; charset=utf-8",
        headers: {
            'RequestVerificationToken': token
        },
        success: function (response) {
            console.log("Success:", response);
            searchProgram().then(() => {
                saveProgramTable();
            }).catch(error => {
                console.log("Search failed:", error);
            });
        },
        error: function (xhr, status, error) {
            console.log("Error:", xhr.status, error);
            console.log("Response Text:", xhr.responseText);
        }
    });

}
```

Figure 28: DeleteProgram ajax call



```
function deletePerson(Person) {
    var data = JSON.parse(decodeURIComponent(Person));

    var token = $('input[name="__RequestVerificationToken"]').val();
    data.__RequestVerificationToken = token;

    $.ajax({
        type: "POST",
        url: "/Home?handler=DeletePerson",
        dataType: "json",
        data: JSON.stringify(data),
        contentType: "application/json; charset=utf-8",
        headers: {
            'RequestVerificationToken': token
        },
        success: function (response) {
            console.log("Success:", response);
            searchPeople().then(() => {
                savePersonTable();
            }).catch(error => {
                console.log("Search failed:", error);
            });
        },
        error: function (xhr, status, error) {
            console.log("Error:", xhr.status, error);
            console.log("Response Text:", xhr.responseText);
        }
    });

}
```

Figure 29: DeletePerson ajax call

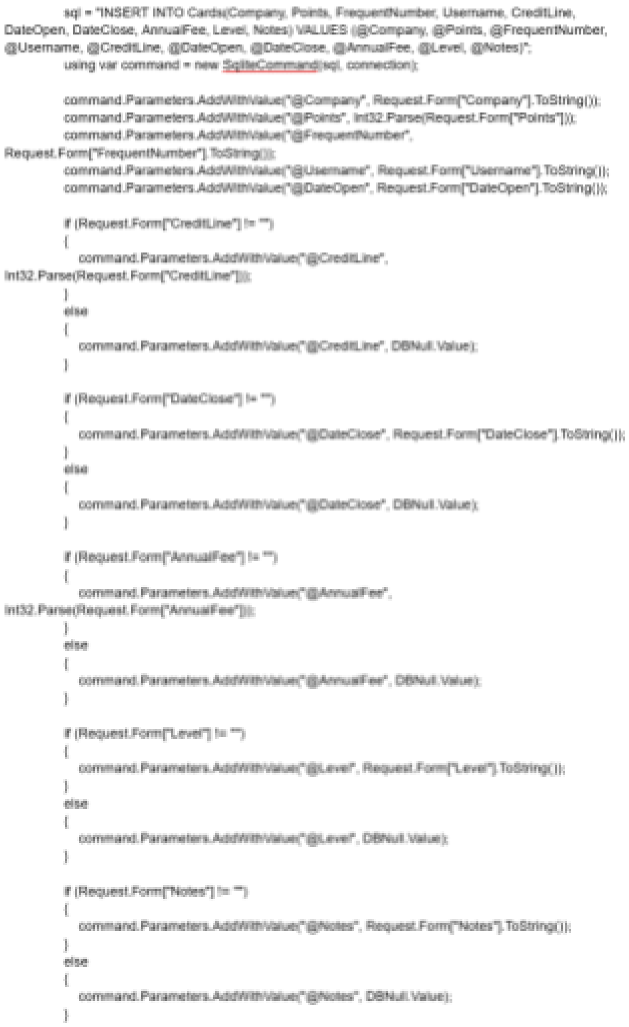


```
function deleteType(Type) {
    var data = JSON.parse(decodeURIComponent(Type));

    var token = $('input[name="__RequestVerificationToken"]').val();
    data.__RequestVerificationToken = token;

    $.ajax({
        type: "POST",
        url: "/Home?handler=DeleteType",
        dataType: "json",
        data: JSON.stringify(data),
        contentType: "application/json; charset=utf-8",
        headers: {
            'RequestVerificationToken': token
        },
        success: function (response) {
            console.log("Success:", response);
            searchType().then(() => {
                saveTypeTable();
            }).catch(error => {
                console.log("Search failed:", error);
            });
        },
        error: function (xhr, status, error) {
            console.log("Error:", xhr.status, error);
            console.log("Response Text:", xhr.responseText);
        }
    });

}
```

Figure 30: DeleteType ajax call

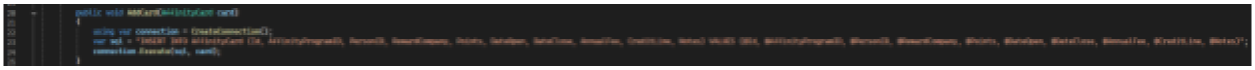
Dapper

Dapper is an object-oriented library that allows for object mapping and more efficient, easier database queries. For comparison, Figure 31 is my previous code of an ON POST add function archived in a google doc, and Figure 32 is the same function with Dapper. As seen by the amount of lines each uses, the Dapper library allows for easier and more efficient coding. For clarification, the archived add card function has different parameters because it was created before the consultations in Appendix B where the move to split the card table into multiple tables was decided on.



```
        sql = "INSERT INTO Cards(Company, Points, FrequentNumber, Username, CreditLine,
DateOpen, DateClose, AnnualFee, Level, Notes) VALUES (@Company, @Points, @FrequentNumber,
@Username, @CreditLine, @DateOpen, @DateClose, @AnnualFee, @Level, @Notes)";
        using var command = new SqliteCommand(sql, connection);

        command.Parameters.AddWithValue("@Company", Request.Form["Company"].ToString());
        command.Parameters.AddWithValue("@Points", Int32.Parse(Request.Form["Points"]));
        command.Parameters.AddWithValue("@FrequentNumber",
Request.Form["FrequentNumber"].ToString());
        command.Parameters.AddWithValue("@Username", Request.Form["Username"].ToString());
        command.Parameters.AddWithValue("@DateOpen", Request.Form["DateOpen"].ToString());

        if (Request.Form["CreditLine"] != "")
        {
          command.Parameters.AddWithValue("@CreditLine",
Int32.Parse(Request.Form["CreditLine"]));
        }
        else
        {
          command.Parameters.AddWithValue("@CreditLine", DBNull.Value);
        }

        if (Request.Form["DateClose"] != "")
        {
          command.Parameters.AddWithValue("@DateClose", Request.Form["DateClose"].ToString());
        }
        else
        {
          command.Parameters.AddWithValue("@DateClose", DBNull.Value);
        }

        if (Request.Form["AnnualFee"] != "")
        {
          command.Parameters.AddWithValue("@AnnualFee",
Int32.Parse(Request.Form["AnnualFee"]));
        }
        else
        {
          command.Parameters.AddWithValue("@AnnualFee", DBNull.Value);
        }

        if (Request.Form["Level"] != "")
        {
          command.Parameters.AddWithValue("@Level", Request.Form["Level"].ToString());
        }
        else
        {
          command.Parameters.AddWithValue("@Level", DBNull.Value);
        }

        if (Request.Form["Notes"] != "")
        {
          command.Parameters.AddWithValue("@Notes", Request.Form["Notes"].ToString());
        }
        else
        {
          command.Parameters.AddWithValue("@Notes", DBNull.Value);
        }
```

Figure 31: Archived add function



```
public void AddCard(AffinityCard card)
{
    using var connection = CreateConnection();
    var sql = "INSERT INTO AffinityCard (Id, AffinityProgramID, PersonID, RewardCompany, Points, DateOpen, DateClose, AnnualFee, CreditLine, Notes) VALUES (@Id, @AffinityProgramID, @PersonID, @RewardCompany, @Points, @DateOpen, @DateClose, @AnnualFee, @CreditLine, @Notes)";
    connection.Execute(sql, card);
}
```

Figure 32: AddCard function

Razor pages

Razor pages were used to develop the web application. This makes the web application more simple to develop. The figure below shows the variables and functions for each model associated with the home page and the login page.

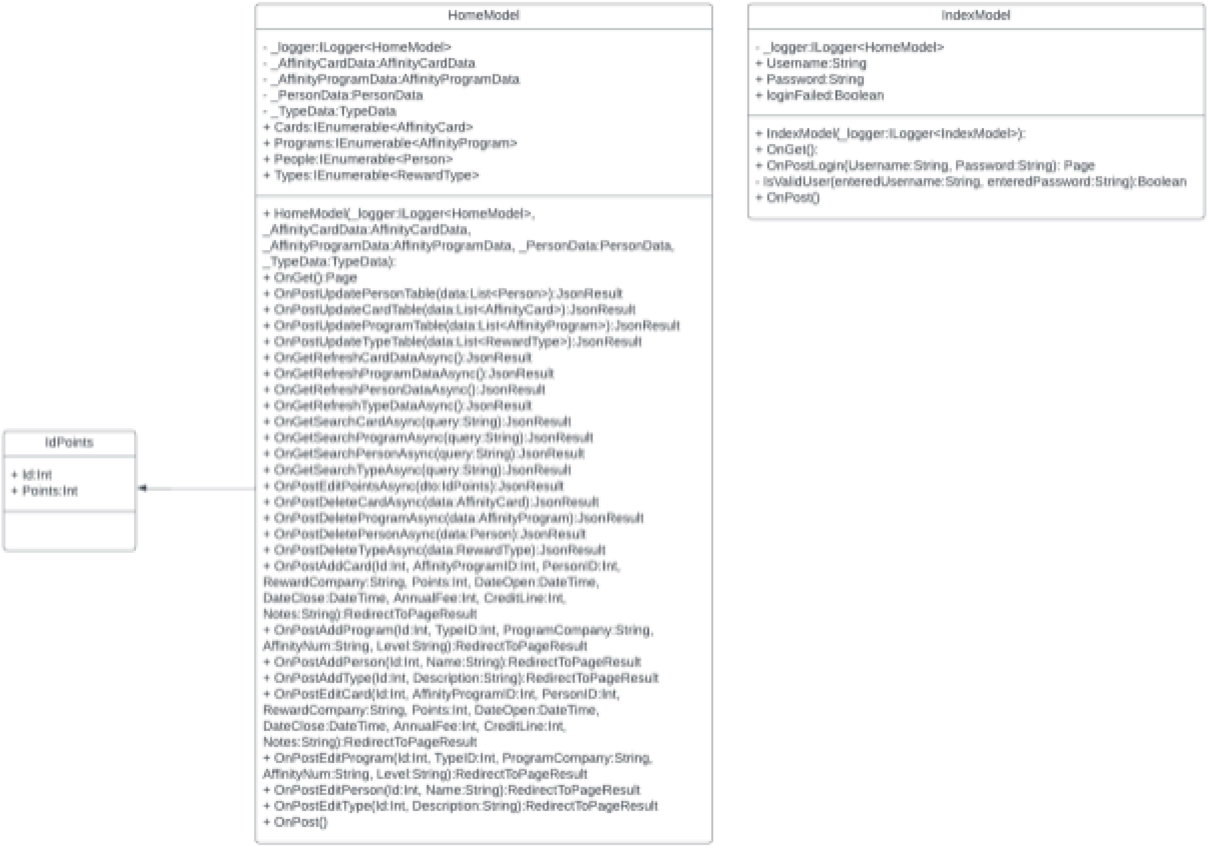

IdPoints
+ Id:Int
+ Points:Int
HomeModel
- _logger:ILogger<HomeModel>
- _AffinityCardData:AffinityCardData
- _AffinityProgramData:AffinityProgramData
- _PersonData:PersonData
- _TypeData:TypeData
+ Cards:IEnumerable<AffinityCard>
+ Programs:IEnumerable<AffinityProgram>
+ People:IEnumerable<Person>
+ Types:IEnumerable<RewardType>
+ HomeModel(_logger:ILogger<HomeModel>, _AffinityCardData:AffinityCardData, _AffinityProgramData:AffinityProgramData, _PersonData:PersonData, _TypeData:TypeData):
+ OnGet():Page
+ OnPostUpdatePersonTable(data:List<Person>):JsonResult
+ OnPostUpdateCardTable(data:List<AffinityCard>):JsonResult
+ OnPostUpdateProgramTable(data:List<AffinityProgram>):JsonResult
+ OnPostUpdateTypeTable(data:List<RewardType>):JsonResult
+ OnGetRefreshCardDataAsync():JsonResult
+ OnGetRefreshProgramDataAsync():JsonResult
+ OnGetRefreshPersonDataAsync():JsonResult
+ OnGetRefreshTypeDataAsync():JsonResult
+ OnGetSearchCardAsync(query:String):JsonResult
+ OnGetSearchProgramAsync(query:String):JsonResult
+ OnGetSearchPersonAsync(query:String):JsonResult
+ OnGetSearchTypeAsync(query:String):JsonResult
+ OnPostEditPointsAsync(dto:IdPoints):JsonResult
+ OnPostDeleteCardAsync(data:AffinityCard):JsonResult
+ OnPostDeleteProgramAsync(data:AffinityProgram):JsonResult
+ OnPostDeletePersonAsync(data:Person):JsonResult
+ OnPostDeleteTypeAsync(data:RewardType):JsonResult
+ OnPostAddCard(Id:Int, AffinityProgramID:Int, PersonID:Int, RewardCompany:String, Points:Int, DateOpen:DateTime, DateClose:DateTime, AnnualFee:Int, CreditLine:Int, Notes:String):RedirectToPageResult
+ OnPostAddProgram(Id:Int, TypeID:Int, ProgramCompany:String, AffinityNum:String, Level:String):RedirectToPageResult
+ OnPostAddPerson(Id:Int, Name:String):RedirectToPageResult
+ OnPostAddType(Id:Int, Description:String):RedirectToPageResult
+ OnPostEditCard(Id:Int, AffinityProgramID:Int, PersonID:Int, RewardCompany:String, Points:Int, DateOpen:DateTime, DateClose:DateTime, AnnualFee:Int, CreditLine:Int, Notes:String):RedirectToPageResult
+ OnPostEditProgram(Id:Int, TypeID:Int, ProgramCompany:String, AffinityNum:String, Level:String):RedirectToPageResult
+ OnPostEditPerson(Id:Int, Name:String):RedirectToPageResult
+ OnPostEditType(Id:Int, Description:String):RedirectToPageResult
+ OnPost()
IndexModel
- _logger:ILogger<HomeModel>
+ Username:String
+ Password:String
+ loginFailed:Boolean
+ IndexModel(_logger:ILogger<IndexModel>):
+ OnGet():
+ OnPostLogin(Username:String, Password:String): Page
- IsValidUser(enteredUsername:String, enteredPassword:String):Boolean
+ OnPost()

Figure 33: Razor pages

Sources

Mark Otto, J.T. (no date) *Bootstrap*, *Bootstrap · The most popular HTML, CSS, and JS library in the world.* Available at: https://getbootstrap.com/ (Accessed: 04 February 2025).

W3schools.com (no date) *(C Sharp)*. Available at: https://www.w3schools.com/cs/index.php (Accessed: 17 June 2024).

Stack Overflow Questions (no date) *Stack Overflow*. Available at: https://stackoverflow.com/questions (Accessed: 17 June 2024).

Word Count: 812