Development Book

Group 1, ISTE.432.01

Fall 2021

Team Members and Roles:

- Siddhesh Periaswami, Front-End
- Ezana Kalekristos, Database Administrator
- Ryan Weiss, Back-End
- Afzal Ali, UI/UX Designer

Background:

Catholic Family Center works to address the barriers preventing people from moving to self-sufficiency in today's environment. Programs are collaborative and focused on systemic change. Clinicians work with high risk clients to develop a safety plan on paper. That plan is printed and handed to the client or, in the case of phone or Zoom encounters, printed and mailed to the client.The plan, being printed, is not interactive and may easily be lost or destroyed or stored somewhere without ready access by the client.

Project Description:

The current system involves printing the safety plan for clients on paper and then handing that plan to the clients. If the client is on the phone or over Zoom, then the safety plan is printed and mailed to the client. This is not ideal because it can be easily lost, destroyed, or not easily accessible to the client. In addition, the paper plan is not interactive and thus may not be as helpful as it could. The main goal of this project is to streamline the process of accessing and editing the safety plan using the clinician's desktop/phone and the client's phone.

Project Requirements:

- Safe and secure storage of safety plans uploaded by the clinician.
- Clinicians should be able to copy/paste the plan into another application.
- The application should ask for access to the contact list to get a friend's phone number for calls
- The users should be presented with choices to create or modify safety plans, and see when a plan was edited and by who.
- The application should have search bar for the users to search for client's safety plans
- The app should allow both clinician and client to enter and delete items since clients may have trouble typing during phone calls.
- Add resources that the clients can access in the app (help lines, on call clinician)
- Signing up. logging in, and logging out

- The clinician should be able to add a client

Business Rules:

- You must login before entering details in the application.
- You must be an authorized user of the system to access (Clinician, Client, Family of Client, etc.)
- When a user signs up for the first time, they should fill up their personal information before proceeding with any other steps.
- Only clients and clinicians can create appointments
- The professionals/agencies that we recommend must be verified to be credible before using with clients
- Only a clinician can invite another clinician to support a client
- The date, time and person who edited a safety plan must be recorded
- All users must be able to logout
- Clinicians must be able to to view all clients in the system
- Has to have full functionality on mobile and desktop

Technologies Used:

Front-End Technologies: React, Javascript

Back-End Technologies: Go

Database: MySQL

UI/UX: Figma

Design Pattern:

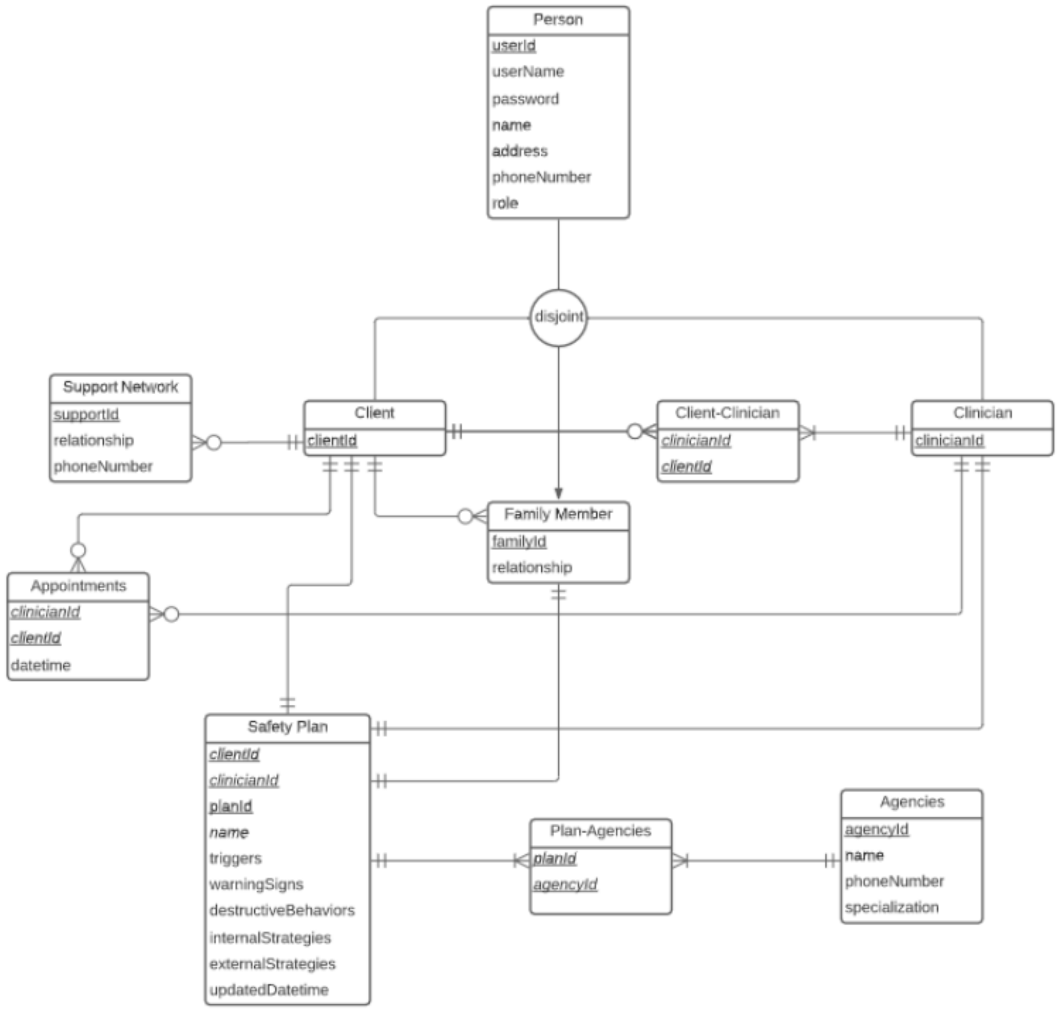
We will incorporate the design pattern MVC into our project for various reasons. One reason would be that it can help enforce our business rules within the application. Within the controller, we can create and enable files that will both create and manage the business logic and the functionality for the application. Another reason is that it would allow us to handle different changes/modifications within the application with minimal changes to other sections of the application. The MVC pattern supports loose coupling, and since we are anticipating later refactoring to code, it will help significantly later in the timeline. This pattern also aids with support for multiple views. Due to the fact that the CFC project is intended to have support for both the Desktop and mobile devices, MVC will allow us to have two different views without needing to adjust the controller or the model at all.

Model: Solely responsible for interacting with the database and handling all data logic. It creates, retrieves, updates, and deletes any data from the database. The Model only communicates with the controller and provides it with whatever information the controller needs.

View: The frontend graphical user interface that the user interacts with. The View only communicates with the controller. The View is used to display the data using the Model object.

Controller: The intermediary between the View and Model. It requests information from the Model and sends it to the View, or it can get information from the View and send it to the Model. This will handle all of the logic for the business rules.

Entity Relationship Diagram:



Layering:

Presentation Layer

- Responsible for formatting and displaying the information of the application. This layer is also responsible for collecting information from the users.

- This layer acts as the user interface of the application where the users interact with the applications. The client and clinician will interact with this layer to communicate with the application.
- This layer will also act as the "View" in our MVC architecture
- Clients will cread, read, update, and delete their warning signs, internal and external coping strategies, emergency contacts, and request immediate help.
- Clinicians would be able to view their clients safety plans and would be notified of any changes the clients make.
- Users(Clients/Clinicians) will be using the UI to login to their respective accounts.

```
const signUp = event => {
        event.preventDefault();
        history.push('/signUp');

    }



    return (
        <div className='loginForm'>
                <img
                    className="logo"
                    src='img.png'
                />

            <div className='container'>
                <h1>SignIn</h1>

                <form>
                    <h5>Email:</h5>
                    <input type='text' value={email} onChange={event => setEm
ail(event.target.value)} />

                    <h5>Password:</h5>
                    <input type='password' value={password} onChange={event =
> setPassword(event.target.value)} />

                    <button type='submit' onClick={signIn} className='signInB
utton'>Sign In</button>
                </form>

                <p>
                    By signing-in you agree to the CFC Conditions & Policies.
Please
                    see our Privacy Notice.
                </p>

                <button onClick={signUp} className='signUpButton'>Sign Up</bu
tton>
            </div>
        </div>
    )
```

The code snippet above shows how the login page is designed. It shows the structure(1 Column) of the login page and the components(Labels, Input Text Fields, Button) of it. The code also shows how these Input Fields are stored in values(React Hook States) and how they can be used while the user hits the submit button. Also one other thing to notice here is that, the user is redirected to the Home page after hitting the login button. The user will be able to view the Safety Plan on the Home page.

Application Layer

- This layer contains the data layer which is responsible for storing and managing the data
- All the data is stored and maintained in a relational database server like MySQL, PostgreSQL or a non relational database like MongoDB, Cassandra
- Data which is stored in Database includes:
  - User details like the username/email, email, password, name, address phone number will be stored in a table in the database.

```
create table cfc.person
(
    userid      serial
        constraint person_pkey
            primary key,
    username    varchar(45) not null,
    password    text        not null,
    firstname   varchar(45) not null,
    lastname    varchar(45) not null,
    email       varchar(45) not null,
    address     varchar(45) not null,
    phonenumber varchar(45) not null,
    role        varchar(45) not null,
    expiration  varchar(65),
    dob         varchar(45)
);
```

In the image above we can see how the table in the database is defined, what are all the column names/attributes defined in the table.

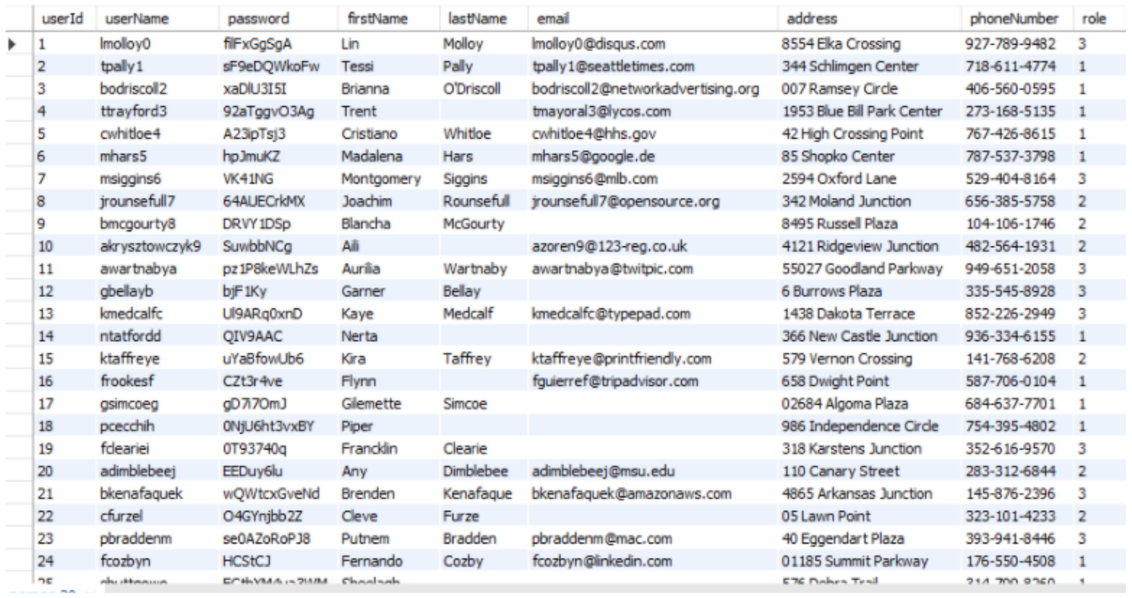
- Other Safety Plan related information like Name, Triggers, Warning Signs, Destructive Behaviors, Internal Coping Strategies External Coping Strategies is stored in a table in the database

Infrastructure Layer

- The DAOs , which will interact with the Facades to load data from the database. This is how the Application layer will communicate with the database. This will be written in Go. There isn't any code yet because we were focused on getting the database created first.

- ClientDAO, which will interact with the ClientFacade. This DAO will add/update the client table, as well as return user information, appointments, support networks, and plans.
- ClinicianDAO, which will interact with the ClinicianFacade. This DAO will add/update clinician information, as well as return clients, plans, and appointments.
- PlanDAO, which will interact with the PlanFacade. This DAO will add/update plans, return plans, and return agencies associated with the plan.
- UserDAO, which will interact with the UserFacade. This DAO will add/update users.
- FamilyDAO, which will interact with the FamilyFacade. This DAO will add/update family information, as well as return plans and clients.

- The facades, which will be interacting directly with the database. These facades will handle the sql statements, and will return the data to the DAOs. Each DAO will have its own facade. The facades will also be written in Go.
- The MYSQL Database.
  - DBConnection is the class used to instantiate the database connection. Every facade will use the DBConnection object to connect.
  - A few inconsistencies in the data are "blanks" rows in some of the "last names" and "email" columns.



userId	userName	password	firstName	lastName	email	address	phoneNumber	role
1	lmolloy0	fiIFxGgSgA	Lin	Molloy	lmolloy0@disqus.com	8554 Elka Crossing	927-789-9482	3
2	tpally1	sF9eDQWkoFw	Tessi	Pally	tpally1@seattletimes.com	344 Schlimgen Center	718-611-4774	1
3	bodriscoll2	xaDlU3I5I	Brianna	O'Driscoll	bodriscoll2@networkadvertising.org	007 Ramsey Circle	406-560-0595	1
4	ttrayford3	92aTggvO3Ag	Trent		tmayoral3@lycos.com	1953 Blue Bill Park Center	273-168-5135	1
5	cwhitloe4	A23ipTsj3	Cristiano	Whitloe	cwhitloe4@hhs.gov	42 High Crossing Point	767-426-8615	1
6	mhars5	hpJmuKZ	Madalena	Hars	mhars5@google.de	85 Shopko Center	787-537-3798	1
7	msiggins6	VK41NG	Montgomery	Siggins	msiggins6@mlb.com	2594 Oxford Lane	529-404-8164	3
8	jrounsefull7	64AUECrkMX	Joachim	Rounsefull	jrounsefull7@opensource.org	342 Moland Junction	656-385-5758	2
9	bmcgourty8	DRVY1DSp	Blancha	McGourty		8495 Russell Plaza	104-106-1746	2
10	akrysztowczyk9	SuwbbNCg	Aili		azoren9@123-reg.co.uk	4121 Ridgeview Junction	482-564-1931	2
11	awartnabya	pz1P8keWLhZs	Aurilia	Wartnaby	awartnabya@twitpic.com	55027 Goodland Parkway	949-651-2058	3
12	gbellayb	bjF1Ky	Garner	Bellay		6 Burrows Plaza	335-545-8928	3
13	kmedcalfc	Ul9ARq0xnD	Kaye	Medcalf	kmedcalfc@typepad.com	1438 Dakota Terrace	852-226-2949	3
14	ntatfordd	QIV9AAC	Nerta			366 New Castle Junction	936-334-6155	1
15	ktaffreye	uYaBfowUb6	Kira	Taffrey	ktaffreye@printfriendly.com	579 Vernon Crossing	141-768-6208	2
16	frookesf	CZt3r4ve	Flynn		fguierref@tripadvisor.com	658 Dwight Point	587-706-0104	1
17	gsimcoeg	gD7i7OmJ	Gilemette	Simcoe		02684 Algoma Plaza	684-637-7701	1
18	pcecchih	0NjU6ht3vxBY	Piper			986 Independence Circle	754-395-4802	1
19	fcleariei	0T93740q	Francklin	Clearie		318 Karstens Junction	352-616-9570	3
20	adimblebeej	EEDuy6lu	Any	Dimblebee	adimblebeej@msu.edu	110 Canary Street	283-312-6844	2
21	bkenafaquek	wQWtcxGveNd	Brenden	Kenafaque	bkenafaquek@amazonaws.com	4865 Arkansas Junction	145-876-2396	3
22	cfurzel	O4GYnjbb2Z	Cleve	Furze		05 Lawn Point	323-101-4233	2
23	pbraddenm	se0AZoRoPJ8	Putnem	Bradden	pbraddenm@mac.com	40 Eggendart Plaza	393-941-8446	3
24	fcozbyn	HCStCJ	Fernando	Cozby	fcozbyn@linkedin.com	01185 Summit Parkway	176-550-4508	1

Exception Handling:

- **Frontend:**

For the current milestone, we have built a login page for the users to access. Users can access the login page, enter their email & password to access their respective home pages. Based on the role of the user whether they are a client/clinician/family member(other), their respective home pages are rendered when the user logs in. There is an option to sign up for new users.

```
const signUp = event => {
        event.preventDefault();
        history.push('/signUp');

    }

const signIn = event => {
        event.preventDefault();

        const credentials = { 'email': email, 'password':password};

        console.log('AJAX')

        console.log(credentials)
        axios({ method: 'post', url: 'http://127.0.0.1:3000/login', data: JSO
N.stringify(credentials)})
        .then((response) => {
            console.log(response.data);
                console.log(response.data['token'])
                setToken(response.data['token'])

                const tokenData = jwt(response.data['token']);
                console.log(tokenData);
                if(tokenData.authorized){
                    if(tokenData.role =='1'){
                        console.log("Yes");
                        console.log(token);
                        console.log(response.data['token'])
                        console.log("Second request")

                        console.log("Bearer "+response.data['token'])

                        let url = "http://127.0.0.1:3000/client"


                        const AuthStr = 'Bearer '.concat(response.data['token
']);

                        axios({ method: 'get', url: 'http://127.0.0.1:3000/cl
ient', headers: { 'Authorization': 'Bearer ' + response.data['token'] } })
                        .then((res) => {
                                        console.log("FINAL", res)
                                        if(res.status  == 200){
                                            history.push({
                                                pathname: '/clientHome',
                                                state: {"Data":res.data, "Tok
en":response.data['token'], "Role":tokenData.role}
                                            })
                                        }
                                    }, (error) => {
                                        console.log("Error"+error)
                                    }
                                );
                    }
                    else if (tokenData.role =='2'){
                        console.log("Clinician data", response.data)
                        history.push({
                            pathname: '/clinicianHome',
                            state: {"Data":[], "Token":response.data['token']
, "Role":tokenData.role}
                        })
                    }
                }
                else{
                    alert("You are not an authorized user");
                }
            }, (error) => {
                console.log("Error"+error)
            }
        );


    }
```

In the two images above, you can see how signIn and signUp are handled as two separate methods. And in these methods, you can see how users are routed to a different homepage based on their role(client/clinician). When the signIn/signUp, the username and email are sent as JSON to the backend api which is shown in the image below.

```
const signIn = event => {
        event.preventDefault();

        const credentials = { 'email': email, 'password':password};

        console.log('AJAX')

        console.log(credentials)
        axios({ method: 'post', url: 'http://127.0.0.1:3000/login', data: JSO
N.stringify(credentials)})
        .then((response) => {
            console.log(response.data);
                console.log(response.data['token'])
                setToken(response.data['token'])

                const tokenData = jwt(response.data['token']);
                console.log(tokenData);
                if(tokenData.authorized){
                    if(tokenData.role =='1'){
                        console.log("Yes");
                        console.log(token);
                        console.log(response.data['token'])
                        console.log("Second request")

                        console.log("Bearer "+response.data['token'])

                        let url = "http://127.0.0.1:3000/client"


                        const AuthStr = 'Bearer '.concat(response.data['token
']);

                        axios({ method: 'get', url: 'http://127.0.0.1:3000/cl
ient', headers: { 'Authorization': 'Bearer ' + response.data['token'] } })
                        .then((res) => {
                                    console.log("FINAL", res)
                                    if(res.status  == 200){
                                        history.push({
                                            pathname: '/clientHome',
                                            state: {"Data":res.data, "Tok
en":response.data['token'], "Role":tokenData.role}
                                        })
                                    }
                                }, (error) => {
                                    console.log("Error"+error)
                                }
                            );
                    }
```

- **Backend**

For the current milestone, we have a route that authenticates a user based on a submitted email and password. If the email and password is missing then a bad request response is sent to the client. If the email and password don't match a user in the database, then an unauthorized response is sent to the client instead. Finally, if the email and password matches a user in the database, that user's data is sent to the client and the role is sent. This will allow the frontend to display different views based on if the user is a client, clinician or family member.

```
type AuthHandler struct {
    Database DB.DatabaseConnection
}

func (ah *AuthHandler) Login(w http.ResponseWriter, r *http.Request) {
    type Login struct {
        Email    string
        Password string
    }
    var logStruct Login
    body := json.NewDecoder(r.Body).Decode(&logStruct)
    if body != nil {
        http.Error(w, body.Error(), http.StatusBadRequest)
        return
    }
    person := Facade.NewPersonFacade(ah.Database)
    pers, _ := person.LoginPersonByEmail(logStruct.Email, logStruct.Password)
    if pers.GetUserID() == 0 {
        http.Error(w, "Bad Login", http.StatusUnauthorized)
        return
    } else {
        tokenString, erro := Auth.GenerateJWT(pers.GetUserID(), pers.GetEmail
(), pers.GetRole())
        if erro != nil {
            http.Error(w, erro.Error(), http.StatusInternalServerError)
            return
        }
        resp := make(map[string]string)
        resp["token"] = tokenString
        b, err := json.Marshal(resp)
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        w.Header().Set("Content-Type", "application/json")
        _, err = w.Write(b)
        if err != nil {
            return
        }
    }
}

// This returns a jwt upon a successful login
func (ah *AuthHandler) signUp(w http.ResponseWriter, r *http.Request) {
    type sign struct {
        Username    string
        FirstName   string
        LastName    string
        Email       string
        Address     string
        Password    string
        PhoneNumber string
        DOB         string
    }
    var signStruct sign
    body := json.NewDecoder(r.Body).Decode(&signStruct)
    if body != nil {
        http.Error(w, body.Error(), http.StatusBadRequest)
        return
    }

    person := Facade.NewPersonFacade(ah.Database)
    newPers := Model.NewPerson(
        signStruct.Username,
        signStruct.Password,
        signStruct.FirstName,
        signStruct.LastName,
        signStruct.Email,
        signStruct.Address,
        signStruct.PhoneNumber,
        "1",
        " ",
        signStruct.DOB)
    userID, err := person.CreateNewPerson(*newPers)
    if err != 1 {
        http.Error(w, "Couldn't Create Person", http.StatusBadRequest)
        return
    } else {
        client := Facade.NewClientFacade(ah.Database)
        clientModel := Model.NewClient(userID)
        client.AddClient(*clientModel)
        resp := make(map[string]string)
        resp["message"] = "Client added to Database"
        b, err := json.Marshal(resp)
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        w.Header().Set("Content-Type", "application/json")
        w.Write(b)
    }
}
```

Performance and refactoring:

- **Backend:**
- Changes to Database
    1. The Person table now has fields for DOB and password expiration
    2. The Clinician table now has a field for referral code, which will be used by the client when they go to create an account on the site. The referral code is used to prevent spam accounts and to automatically assign that clinician to the client.
- Updated routes for login and for clinicians to get clients
    1. JWT Authentication tokens implemented for users
- Updated facades and DAOs for more functionality

1. Mostly just adding the ability to get data from multiple tables within a facade (example: Able to get Safety Plans from the ClientFacade via clientID) -The ER Diagram was also updated (see above)

- **Frontend:**
- The frontend allows only authorized users to sign in
- Modified Client's and Clinician's homepage to list all the options a user can do upon landing on the home page, also included a navbar.
- Clients can now view their profile by clicking on the View My Profile button on their homepage
- Clinicians can view/search for users by clicking on View Users button on their homepage
- Refactored the entire frontend code to separate View components and the actual components which contain the data. Following this strategy made the code look cleaner, readable and easily understandable. Earlier two other components were grouped with the View component which made the code look very messy and cluttered. In the code snippet below, we can see one example of how the code has been restructured.

```
return (
        <>
            <Header header="Client's Homepage"/>
            <div style={{textAlign:"center", marginLeft:"auto", marginRight:"
auto"}}>
                <h4>Welcome {firstname}</h4>
                <h5>What would you like to do today?</h5>
                <button onClick={viewProfile} class="myButton">View my profil
e</button>
                <button class="myButton">View my safety plan</button>
                <button class="myButton">Add emergency contact</button>
            </div>

        </>
    )
```

This is the code used to render the Client's homepage. Notice how the Navbar/Header is split into a component of it's own and being used(in the second line inside return method) in this View component. A similar code structure will be noticeable in all the other components as well. Another example of refactored code:

```
<Header header="Clients"/>


        <div style={{textAlign:"center"}}>

            <h3>Clients</h3>
            <Search/>
            <div className="DivWithScroll">
            {usersArray.map((user) => (
  <Card style={{marginLeft:"auto",marginRight:"auto", marginTop:"3%", width:
```

```
'18rem' }}>
                    <Card.Body>
                        <Card.Title>{user.FirstName + user.LastName}</Card.Ti
tle>
                        <Card.Link href="#">Safety Plan</Card.Link>
                        <Card.Link href="#">Profile</Card.Link>
                    </Card.Body>
                </Card>
                ))}
            </div>
```

This is the view of the View Users page. Notice how the Search bar is defined as its own component and being called here in this component to Search and filter users/clients. Also notice how the same Header component is being reused in this case.

The backend code is also structured/refactored in a way that the code is easy readable, understandable.

Refactoring the code improves the overall structure of the code, it makes the code more efficient and highly cohesive. It also helps in making the application loosely coupled, hence improving the overall performance of the application.

Testing:

- **Frontend:** Used Jest library in React to perform testing on the frontend. All the following tests use a JWT Token which is hard coded, the JWT token expires after few minutes. To run the tests successfully, a valid JWT Token should be used in the code. Following snippet shows the output window when the tests are run successfully. `Test Suites: 3 passed, 3 total     Tests:       3 passed, 3 total     Snapshots:   0 total     Time:        4.103 s     Ran all test suites related to changed files.`

    - Testing Login call to the backend from frontend: The following code can be found in Login.test.js flie in the src directory of frontend.

```
test('Test login route', async() => {
    var mock = new MockAdapter(axios);
    const data = {
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdXRob3JpemVk
Ijp0cnVlLCJlbWFpbCI6ImNsaWVudEBnbWFpbC5jb20iLCJleHAiOjE2Mzc4MTIwNTIsInJ
vbGUiOiIxIiwidXNlcklEIjoxMDA1fQ.cxX5sESNtFB9WG3D43XnboWSrxYtbJ8dzYwhKp1
Y9mM"
    };
    mock.onPost('http://127.0.0.1:3000/login').reply(200, data);

    const resp = await Login_Test({"email": "client@gmail.com", "passwo
rd": "cliepassword"});

    expect(resp).toEqual(data);

});
```

In the code snippet above, observe how the mock response object is created as data. Notice how a post call is made using mock.post and the expected result is passed as a parameter to the reply method. And then the Login_Test method is called from a different file and that triggers an API call to the backend to get the response back. And then both the responses are comapred and if they are equal, they pass the test.

- Testing Safetyplan call to the backend from frontend: The following code can be found in Client_SafetyPlan.test.js flie in the src directory of frontend.

```
test('Test for safetyplan of a client', async() => {
    var mock = new MockAdapter(axios);
    const token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdXRob3JpemV
kIjp0cnVlLCJlbWFpbCI6ImNsaWVudEBnbWFpbC5jb20iLCJleHAiOjE2Mzc4MTI5MDAsIn
JvbGUiOiIxIiwidXNlcklEIjoxMDA1fQ.bmqBPT73VCE4NT0bIoN_jfrYwDBxQgwEVeYpo3
UgtuE';
    const data = [
        {
            "SafetyID": 3,
            "Triggers": "Puppies",
            "WarningSigns": "Manic Behavior",
            "DestructiveBehaviors": "N/A",
            "InternalStrategies": "Playing with cats",
            "UpdatedClinician": 341,
            "UpdatedDatetime": "2021-11-16T22:51:29Z",
            "ClientID": 334,
            "ClinicianID": 341
        }
    ];
    mock.onGet('http://127.0.0.1:3000/client/safetyplan',{'Authorizatio
n': 'Bearer ' + token}).reply(200, data);

    const resp = await Client_SafetyPlan_Test(token);

    expect(resp).toEqual(data);

});
```

In the code snippet above, observe how the mock response object is created as data. The data consists an object containing the safety plan information for that particular client. The JWT Token is hard coded, it changes everytime it expires. Notice how a get call is made using mock.get and the expected result is passed as a parameter to the reply method. And then the Login_Test method is called from a different file and that triggers an API call to the backend to get the response back. And then both the responses are comapred and if they are equal, they pass the test.

- Testing CinicianUsers call to the backend from frontend: The following code can be found in ClinicianUsers.test.js flie in the src directory of frontend.

```
test('Test for all clients of a clinician', async() => {
    var mock = new MockAdapter(axios);
    const token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdXRob3JpemV
kIjp0cnVlLCJlbWFpbCI6ImNsaW5pY2lhbkBnbWFpbC5jb20iLCJleHAiOjE2Mzc4MTM2MD
UsInJvbGUiOiIyIiwidXNlcklEIjoxMDA0fQ.Jt_QsOGbA6SWaKiqK6Jr4VEDnIeENlp-KU
SMpPtX-sw';
    const data = [...];
    mock.onGet('http://127.0.0.1:3000/clinician/clients',{'Authorizatio
n': 'Bearer ' + token}).reply(200, data);

    const resp = await Clinician_Users_Test(token);

    expect(resp).toEqual(data);

});
```

In the code snippet above, observe how the mock response object is created as data. It consists a list of clients as an array. The JWT Token is hard coded, it changes everytime it expires. Notice how a get call is made using mock.get and the expected result is passed as a parameter to the reply method. And then the Login_Test method is called from a different file and that triggers an API call to the backend to get the response back. And then both the responses are comapred and if they are equal, they pass the test.

Packaging:

- Please proceed to this link - https://github.com/ezana234/DAD-Project to get access to all the project files.
- Use the following command to clone the repository in your system - git clone https://github.com/ezana234/DAD-Project
- The CFC directory is the directory you want to use, it has both frontend and backend related files in directories called frontend and backend respectively.
- The frontend direcotry has a public directory which contains a index.html file. This file is the build file you should use to deploy. You can recreate the build file using the command npm run build inside the frontend directory.
- And if you want to run the app(frontend and backend) in your system, go to the last section of this file(How to run the application). That section demonstrates how to run the application in your system.

Timeline:

- Milestone3 (Layering) - October 8, 2021
- Milestone4 (ExceptionHandling) - October 22, 2021
- Milestone5 (Refactoring) - November 5, 2021
- Milestone6 (Testing) - November 19, 2021
- Milestone7 (Packaging) - November 29, 2021
- Finalized Project Code - December 6, 2021

Figma wireframe design link

https://www.figma.com/file/UsuDPNGwtqjoFeqcGgmUxU/cfc_wireframe?node-id=0%3A1

How to run the application

Backend:

- First, navigate to `DAD-Project/CFC/backend`
- Run `go build main.go`
- Run `go run main.go`. The server runs on http://localhost:3000

Frontend:

- First, run `npm install`
- Then, run `npm start` to run the application in development mode. You can view it by navigating to http://localhost:3001

Test User Accounts:

There are a few test user accounts that you can log in with.

Clinicians:

Danny Michaels:

- Email: *dmichaels@cfc.com*
- Password: *dmpassword*
- Referral: *DMICHCFC*

Clients

Bobby Tarantino

- Email: *btarantino@gmail.com*
- Password: *btpassword*

Slim Shady

- Email: *sshady@gmail.com*
- Password: *sspassword*