Project Report - Web-Based behavioral Group Therapy Application
By Team Code-Blooded

Team members: Sudeepta Bal - 801455628, Chetan Kapadia - 801438508, Rucha Tatawar - 801420899, Sanjyot Sathe - 801426514 and Nidhi Shah - 801420572

Github repo link:

https://github.com/sbal2911/SPL_Final_project

Introduction – Inspiration

We noticed that the tools used for **managing group therapy are often outdated and scattered**, which **makes it hard for both therapists and patients to stay organized**. This project was motivated **by the need for a more modern, reliable system that handles these sessions smoothly while respecting the sensitivity of the data involved. This project was driven by the inspiration to make use of a language that fosters cleaner, safer and more predictable code. Fsharp represents a style of programming where data flows through clear transformations, functions act predictably and unexpected side effects are minimized.** This is what motivated us to explore more about this language. Emphasis on immutability, robust types and pattern matching goes a long way in taking away many common programming errors. We aimed to create a system where core functionalities, including input validation, workflow management and operation processing, appear structured and consistent. Using Fsharp, we wanted to build an application with logic that was not only functional but easier to maintain, understand and extend.

Language Requirements

The paradigm of the language

Fsharp encourages a style of programming called functional-first language. **Where functions, immutable values and predictable transformations are the primary way of expressing logic. The approach reduces hidden side effects and makes it easier to break down complex computations into manageable, understandable pieces. While Fsharp also supports object-oriented and imperative styles of programming, its functional features are at the heart of how most developers use the language.** Concepts such as pattern matching, function composition, discriminated unions and robust type inference allow developers to express their workflow and edge cases clearly. This setup is better suited to tasks such as data modeling, validation execution and the handling of branching logic in a controlled manner.

Some historical account of the evolution of the language and its antecedents

Fsharp, created by Don Syme (a computer scientist based out of Australia), is a member of the Meta family of languages and draws a great deal of inspiration from OCaml, which combines academic research in type theory with practical programming techniques. It was introduced as a language on the .NET platform to combine functional programming with the wide-ranging tools and libraries already available in the .NET ecosystem. Over the years, Fsharp has developed into a multi-purpose programming language suited for data processing, financial modeling, cloud computing and backend development. In class with functional programming, Fsharp's design promotes robust static typing, highly expressed data structures and succinct syntax. It also has a high degree of interoperability with C# and the rest of the .NET family. As such, Fsharp is well-suited to certain use cases where the functional programming is not enough, but clarity, reliability and maintainability from strategic use of functional programming may be a great asset to the project.

The elements of the language: reserved words, primitive data types, structured types

Fsharp provides a clean and expressive set of language elements that support its functional-first design philosophy. The language includes a collection of **reserved keywords** such as **let, type, match, with, module, open, if, elif, else, for, while, try, and with**. These keywords appear frequently in our project to define functions, model data, perform branching logic, handle exceptions, organize modules, and interact with the .NET ecosystem. Their consistent behavior across contexts contributes to the predictability and regularity that Fsharp emphasizes.

Fsharp supports a wide range of **primitive data types**, many of which appear throughout our group therapy API:

- **int**: used for numeric counts such as totals of events or validation flags
- **string**: used extensively for user fields, event names, categories, and messages
- **bool**: used for validation, conditional checks, and for controlling branching pathways
- **float/double**: occasionally used for calculations or numeric transformations
  These primitive types integrate seamlessly with .NET built-in methods (e.g., String.IsNullOrWhiteSpace, Math.Abs, Equals), allowing our API to blend Fsharp expressiveness with existing framework capabilities.

In addition to primitives, Fsharp provides several **structured types** that are central to how we built the application.

- **Records** were used to define users, events, and RSVPs. They offer immutable, thin data containers that map directly to MongoDB documents and improve readability.

- **Discriminated Unions (DUs)** allow modeling domain concepts such as user roles or validation results in a safe and expressive manner.

- **Anonymous records** let the API construct lightweight JSON responses without defining new types.

- **Arrays and Lists** appear in validation logic, filtering operations, and temporary storage. Arrays support indexed access, while lists support more functional, immutable workflows.

- **Structs** were used only when a lightweight value type made sense, such as returning small summary objects.

Altogether, these language elements gave the project a strong foundation for modeling the domain of therapy sessions, users, and events in a predictable and maintainable way.

A description (in some form) of the syntax of the language

Fsharp uses a functional, expression-driven style where most pieces of code return values. Its syntax minimizes punctuation and relies on indentation, which makes the code look cleaner and easier to follow. In our project, simple **let** bindings helped create straightforward, step-by-step expressions, and the |> pipeline operator lets us pass data through a sequence of transformations in a very readable, left-to-right manner.

Pattern matching (**match … with**) provided a powerful way to express validation logic, error handling, and domain branching without the verbosity typically seen in imperative languages. Its syntax allowed us to describe all possible cases for a given input explicitly, improving safety and correctness.

Function definitions tend to be short—often just one line—and features like partial application made the event-filtering and user-processing logic compact without losing clarity. We also organized our code into modules and namespaces, which use indentation instead of braces for structure. This fits well with Fsharp's functional style and helps keep the overall codebase clean and easy to navigate.

The basic control abstractions of the language (loops, conditional controls, etc.)

Although F# is functional-first, the language provides a full range of control abstractions that we used throughout the development of our API.

Conditional expressions (**if/elif/else**) appear in many parts of the project, including event validation, categorization logic, and determination of status messages. Because conditionals return values rather than just performing actions, they fit naturally into F#'s expression-oriented design and helped keep logic compact.

Loops such as **for** and **while** are available when iterative control is necessary. Our project uses **for** loops in validation sequences (e.g., counting boolean flags), but many traditional loop-heavy tasks were instead expressed through functional list operations (List.map, List.filter, List.iter), which better align with the immutable nature of F#.

Lambda expressions (fun x -> ...) were used extensively in list filtering and API workflows, particularly when transforming event data or filtering categories. They offer a clear and lightweight way to embed small behaviors directly within a pipeline.

Exception handling uses the try ... with syntax, which enables our controllers to catch database access issues, handle serialization failures, and return meaningful error responses. The syntax integrates cleanly with F#'s functional structure while still providing access to standard .NET exception mechanics.

Finally, asynchronous workflows (async { ... }) served as F#'s abstraction for concurrency. These workflows made it easy to perform non-blocking operations in our API while maintaining a declarative structure. They allowed the system to handle multiple simultaneous therapy session interactions, message operations, and database calls without compromising readability or maintainability.

How the language handles abstraction (including functions, procedures, objects, modules, etc.)

F# approaches abstraction in a very practical and down to earth way . **Since the language is built around functions, most of the code naturally ends up being written as small pieces that do one thing clearly. These functions can be passed around or combined without much ceremony , so it becomes easier to solve bigger problems by stitching together many tiny, predictable steps. A lot of everyday logic in F# is introduced through simple "let" bindings, which keeps the flow of the program easy to follow. For organizing code on a larger scale ,**

F# relies on modules and namespaces that act like containers; they help separate different parts of the project and keep internal details hidden when they don;'t need to be exposed. Although the language is heavily functional, it doesn't shut out object-oriented programming. Developers can still define classes, interface, or object expressions when it suits the task or when the .NET environment calls for that style. On top of that F# offers records, discriminate unions, and strong type inference, making data modeling much cleaner and less repetitive. Because of these tools, writing abstractions in F# feels less like wrestling with the language and more like shaping the program around the problem itself .

An evaluation of the language's writability, readability, and reliability

When we look at F# through the lens of the design principles the language does quite well in terms of writability, readability and reliability . Its readability comes from the fact that the syntax is very clean and consistent , so there isn;t much extra clutter to get in the way of understanding what the code is doing. Chapter 2 talks about this idea as regularity , where language features behave the same way across different situations instead of surprising the programmer. F# sticks closely to this idea, features like pattern matching and its expression oriented style follow predictable rules , making the flow of a program much easier to follow.
On the writability side, F# lets developers express complicated logic without needing a lot of boilerplate . Features such as type inference, higher-order functions, immutability, and its functional structure help keep code short but still clear. F# expressiveness and conciseness help programmers work more efficiently, since shorter, clearer code is easier to write and modify. F# fits this nicely by letting developers describe their ideas directly instead of forcing them to fight with extra language syntax.
F#'s type system catches many errors early in the development process and its focus on immutable data helps avoid issues related to accidental changes or shared state problems that often lead to bugs in other languages. Altogether, F# stays close to the principles emphasized in the chapter- regularity, expressiveness , strong typing and maintainability ,making it a solid and trustworthy choice for building reliable software systems.

Major Strengths and Weaknesses of F# language

Strengths: F# supports the Functional way of programming, that results in developing a very understandable logic and less unexpected consequences especially when dealing with huge applications. The language is also known for its strict static typing mixed with type inference, that helps the developers to write compact code and still be on the profit side with the help of errors being detected early. F# executes using the .NET platform, because

of which it can leverage the same libraries, tools and runtime environment that are accessible to C# and remaining .NET languages, hence providing solid groundwork. The syntax used by F# is very minimal and communicative, that helps in lessening the boilerplate code and makes difficult operations super easy to understand and maintain. F# takes care of data transformations and analytical workloads in a very organized way, which in turn proves to be very helpful for domains like scientific computing or finance or any applications that deal with a lot of data. F# facilitates the provision of immutable data structures by default, that proves to be beneficial for the developers to write safer simultaneous programs without having to deal with many problems that come up with shared state. Matching patterns is one of the great feature of F# that provides us with a clean and organized way to tackle different cases without the need of intricate conditional statements.

Weaknesses: With comparison to the mainstream languages, F# maintains a very limited community and ecosystem that gives us an idea about the difficulty we face in locating any ready-made instances, libraries and some learning resources. Speaking of the industry, adoption of F# is still very limited, so teams might have to think before selecting it and developers might get very fewer job opportunities that requires F# to be the primary skill. Another weakness is the support for tooling which is definitely on track for improvement, but still it is not as polished as we have it for C# more specifically in areas such as help required for debugging or refactoring that has been automated or any code analysis done on an advanced level. Developers who actually are used to code in object-oriented languages need some time in learning F# concepts like immutability, recursion, functions of higher order as these concepts might seem new for them. Few .NET libraries are used primarily for C# and if we try applying them in F# too, it may lead to longer, more complex code in certain examples. For some cases like quick scripting tasks, it still cannot match languages like Python which are simple and very flexible and hence preferred more for lightweight automation and trial-and-error coding.

An overview of the F# programs

The F# programs combined together actually form a whole ASP .NET Core Web API system that actually is the supporting factor behind the event management, user management and RSVP handling. The F# controllers are responsible for interacting with the MongoDB to make way for the clients to create, remove, update and fetch documents for users, events and the RSVPs. The EventController program takes care of the event-related operations such as retrieving all the events, validating and verifying the new event data, updating the details of the event and deleting events. The RSVPController program is responsible for the creation and deletion of the RSVP records by the clients, while also fetching all the RSVP

records from the database. In a much similar way, the UserController takes care of user creation, login attempts and also the retrieval of all the users who are registered. **The Event.fs, User.fs and Rsvp.fs are the supporting files that are responsible for defining the strongly typed data models that get directly mapped to the MongoDB documents. The API server is initialized by the program.fs file, which also does the action of registering controllers, enabling Swagger UI, configuring the services that are required, and finally begins the pipeline, making sure that all the controllers are indeed discoverable and functional.**

A brief discussion of what the language features are highlighted in our code

Overall, our code demonstrates variety of noticeable F# language characteristics that are responsible for enhancing clarity and lessening the boilerplate. A thorough and rigorous usage of F# record types are being included, that is required for the provision of concise and readable data models for events, RSVPs and the users. F#'s functional way is being properly highlighted by the controllers through the medium of list processing expressions, lambda functions and the piped transformations, that is responsible for data manipulation and clean filtering. API replies are constructed rapidly by the help of anonymous records without the need for defining additional types. F#'s smooth interoperability with the .NET system, has been well showcased in our code. MongoDB driver operations, logging functionalities and ASP .NET core characteristics combines in a more natural way into the F# syntax. Implicit immutability is depicted in almost the majority portion of our code, the explicit style of immutability is shown only where it is required and essential, keeping in mind the clarity and the safety part. **The attributes like [<ApiController>], [<HttpPost>], and BSON mapping features also demonstrate F#'s effectiveness with external systems, while still maintaining the minimalistic syntax that is highly expressive too.**

How the F# language made the programs easy/hard to implement

F#'s motive of making the execution of the programs pretty easier primarily was successful because of its strong type system, functional characteristics and its minimalistic syntax. The effort that would require to explain the data structures was reduced to a huge extent with the help of F#'s record definitions, that enables the shortening of the model files that in turn make them readable. Functional operations such as lambda expressions or the List.filter, supported in processing categories, validations and the series of events in a more precise way than the traditional looping structure. Developers were able to create ordered and organized JSON replies with the help of anonymous records, without the need of any extra

classes. F#'s pipelining operator helped in improving the readability aspect by keeping linear and expressive operations. .NET libraries are well integrated with F#, due to which the MongoDB, ASP.NET Core and the JSON serialization were made pretty straightforward.

There were few aspects of F# that were a bit hard like the construction of the MongoDB filters that becomes more verbose in F# because of the library being designed around C# expression trees, and also leveraging the mutable state needs more external handling as F# has the feature of immutability by default. However, on a broader level, these obstacles are minor in comparison with the advantages as overall, the design of the language made the programs more clean, easier to maintain and straightforward execution.

Sample run of a program in the F# language and/or platform

- Data Types and Built-in functions

A. Strings + Built-in String Methods

In our code EventController.fs, there is a section on filtering events leveraging String.IsNullOrWhiteSpace

Code:

```
let categories = events |> List.filter (fun e -> not (String.IsNullOrWhiteSpace(e.category)))
```

The above line leverages the F#'s List.filter function mixed with a .NET built-in method String.IsNullOrWhiteSpace, portraying a direct instance of data manipulation on the string data type. Basically, this code is meant for filtering out events in which the category field is either a whitespace or empty. This portrays F#'s way of processing functional list and the .NET interop. The line also tells us how the integration of F# is carried out smoothly with other C# string utilities.

B. Booleans and Boolean operations

In our code EventController.fs, there is a section on validation flags array

Code:

```
let validations = [| true; false; true; true |]
        logger.LogInformation("CreateEvent validations: {@validations}", validations)
```

```
// Use for loop to count valid flags
let mutable validCount = 0
for isValid in validations do
    logger.LogInformation("CreateEvent isValid: {@isValid}", isValid)
    if isValid then validCount <- validCount + 1
```

This section leverages the boolean values like true and false, and uses an if condition for the purpose of its evaluation. The validations array helps in storing the logical conditions. This small portion demonstrates to us the way how booleans support logic flow in an actual backend API.

C. Integers and Numeric Built-in Methods (Counting, Comparison)

In our code, EventController.fs , we can observe the below lines:

Code:

```
let totalEvents:int = allEvents.Count
let isEqual = totalEvents.Equals(100)
```

In this section, integer values are calculated first and then leveraged the built-in .Equals() method for the purpose of comparison. This code evaluates the number of events that are present in the database and compares it with a target count of 100. The usage of **.Count** and **.Equals** represents the built-in numeric manipulation functions.

- Data Structures (Array, List, Struct) & Control Structures (For, If-Else, Lambda)

A. Arrays

In our code, EventController.fs , we can observe the below lines:

Code:

```
let validations = [| true; false; true; true |]
```

This is an instance of a classic F# array literal. The array here stores the boolean values leveraged for the logic of validation. Arrays in F# encourages fast indexing mechanism and are suitable for simple fixed-size collections like the ones stated above

B. Lists

In our code, RsvpController.fs and EventController.fs , we can observe the below lines:

Code:

```
static let mutable events = []
let categories = events |> List.filter (fun e -> not
(String.IsNullOrWhiteSpace(e.category)))
```

This is an F# list leveraged for the purpose of storing the event object temporarily. The **List.filte**r call leverages the Lambda function satisfying the criteria for the functional way of control structures. List are unchangeable if they are not being replaced, which actually fulfils the F#'s functional nature.

C. Struct

In our code, EventController.fs , we can observe the below lines:

Code:

```
type EventSummary =
   val mutable Name: string
   val mutable IsValid: bool
   new(name, isValid) = { Name = name; IsValid = isValid }
```

EventSummary is a very lightweight value-type container leveraged to return the summarized information about an event. For the purpose of efficiency and idealism in case of small grouped data, structs in F# serves the need. This struct is later included into the JSON reply showcasing its practical worth.

D. For loop

In our code, EventController.fs , we can observe the below lines:

Code:

```
let mutable validCount = 0
        for isValid in validations do
           logger.LogInformation("CreateEvent isValid: {@isValid}", isValid)
           if isValid then validCount <- validCount + 1
```

This is an example of direct F# for loop over an array. The loop actually iterates over all the boolean validation flags and finally updates a counter variable. It also leverages if-else logic within the loop, showcasing both control structures parallely. This instance has the actual utility of the backend in verifying the event data submitted by an user.

E. If-Else Conditional Structure

In our code, EventController.fs , we can observe the below lines:

Code:

```
// Use if-else to determine status
        let statusMessage =
            if categories.Length > 0 then
                "Event accepted"

            else
                "Event rejected: missing category"
```

This section determines the conditional branching logic. This logic states whether a valid category is there for an event or not, and accordingly creates an appropriate reply message. The simplicity of F#'s if-else syntax is well depicted in this code, especially when mixed with checks concerning list-length.

F. Lambda Functions

In our code, EventController.fs , we can observe the below lines:

Code:

```
events |> List.filter (fun e -> not (String.IsNullOrWhiteSpace(e.category)))
```

The lambda functions get applied during the filtering of the list to understand whether there is a valid category for an event or not. This functional style is an advantage of core F# concepts. It showcases the one-line logic without the need for boilerplate that in turn is very clean and expressive.

- Exception Handling

A. Try-With Exception Handling

In all our controllers, EventController.fs, UserController.fs and RsvpController.fs , we can observe the below lines:

Code:

```
try
    let eventsCollection = database.GetCollection<GetEvent>("events")
```

```
with ex ->
    let message: string = "GetEvents:Error:" + ex.Message
    logger.LogError(ex, message)
    base.StatusCode(500, message)
```

This block of code portrays the exceptional handling procedure in F#. It shows us the way to handle any database or serialization errors during the process of HTTP request. The handler logs the error and returns a 500 reply to the client.

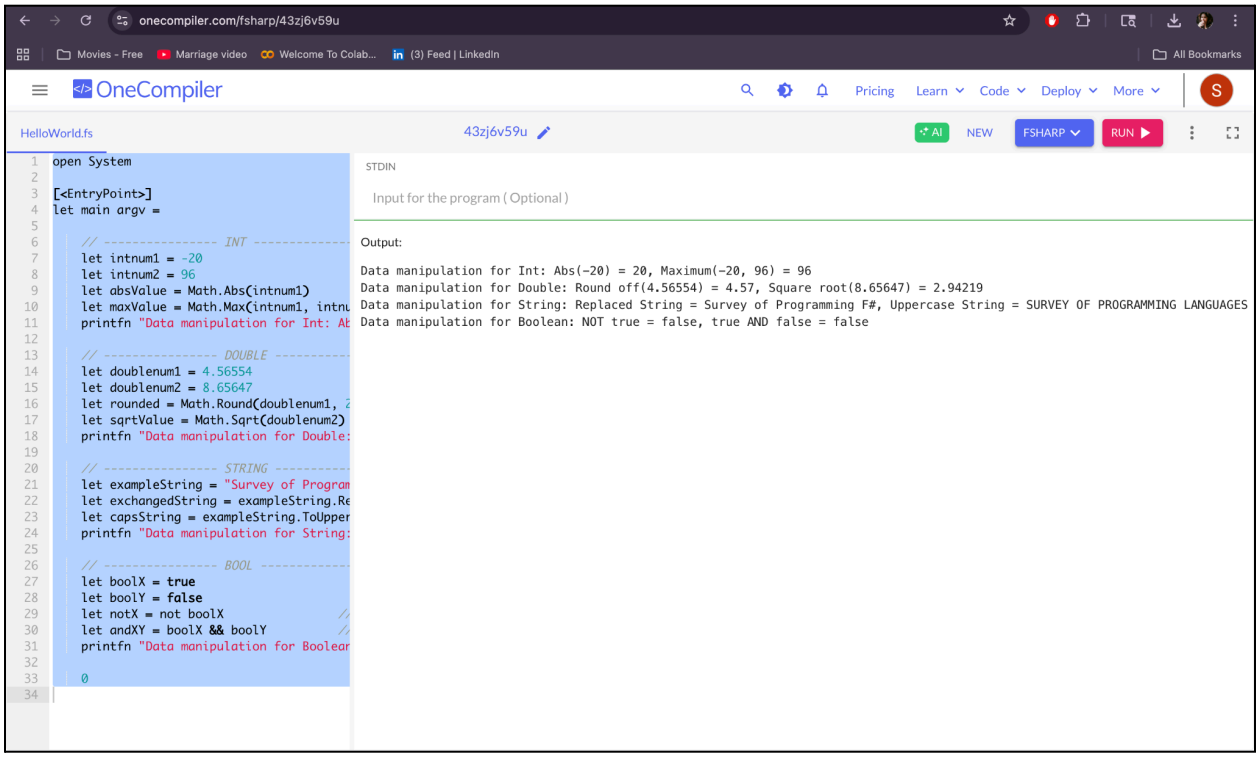
Apart from our project's coding logic, please refer to the below screenshots as well to see the exact output for all these features discussed above with the help of small easy programs and their outputs:

- 4 data types(example: int, float, string, boolean) and showcase 2 in built methods(per data type total =8) showcasing data manipulations (example :abs, average string replace)

Code:

```
open System

[<EntryPoint>]
let main argv =

    // ---------------- INT ----------------
    let intnum1 = -20
    let intnum2 = 96
     let absValue = Math.Abs(intnum1)              // Built-in method 1: absolute value (manipulates data)
     let maxValue = Math.Max(intnum1, intnum2)      // Built-in method 2: maximum (manipulates data)
      printfn "Data manipulation for Int: Abs(%d) = %d, Maximum(%d, %d) = %d" intnum1 absValue intnum1 intnum2 maxValue

    // ---------------- DOUBLE ----------------
    let doublenum1 = 4.56554
    let doublenum2 = 8.65647
    let rounded = Math.Round(doublenum1, 2)     // Built-in method 1: round to 2 decimals
    let sqrtValue = Math.Sqrt(doublenum2)       // Built-in method 2: square root
     printfn "Data manipulation for Double: Round off(%.5f) = %.2f, Square root(%.5f) = %.5f" doublenum1 rounded doublenum2 sqrtValue
```

```
// ---------- STRING ----------
let exampleString = "Survey of Programming Languages"
    let exchangedString = exampleString.Replace("Languages", "F#") // Built-in method 1: replace substring
let capsString = exampleString.ToUpper()                // Built-in method 2: convert to uppercase
    printfn "Data manipulation for String: Replaced String = %s, Uppercase String = %s" exchangedString capsString

// ---------- BOOL ----------
let boolX = true
let boolY = false
let notX = not boolX                // Built-in method 1: logical NOT
let andXY = boolX && boolY          // Built-in method 2: logical AND
  printfn "Data manipulation for Boolean: NOT %b = %b, %b AND %b = %b" boolX notX boolX boolY andXY

0     // Successful execution
```

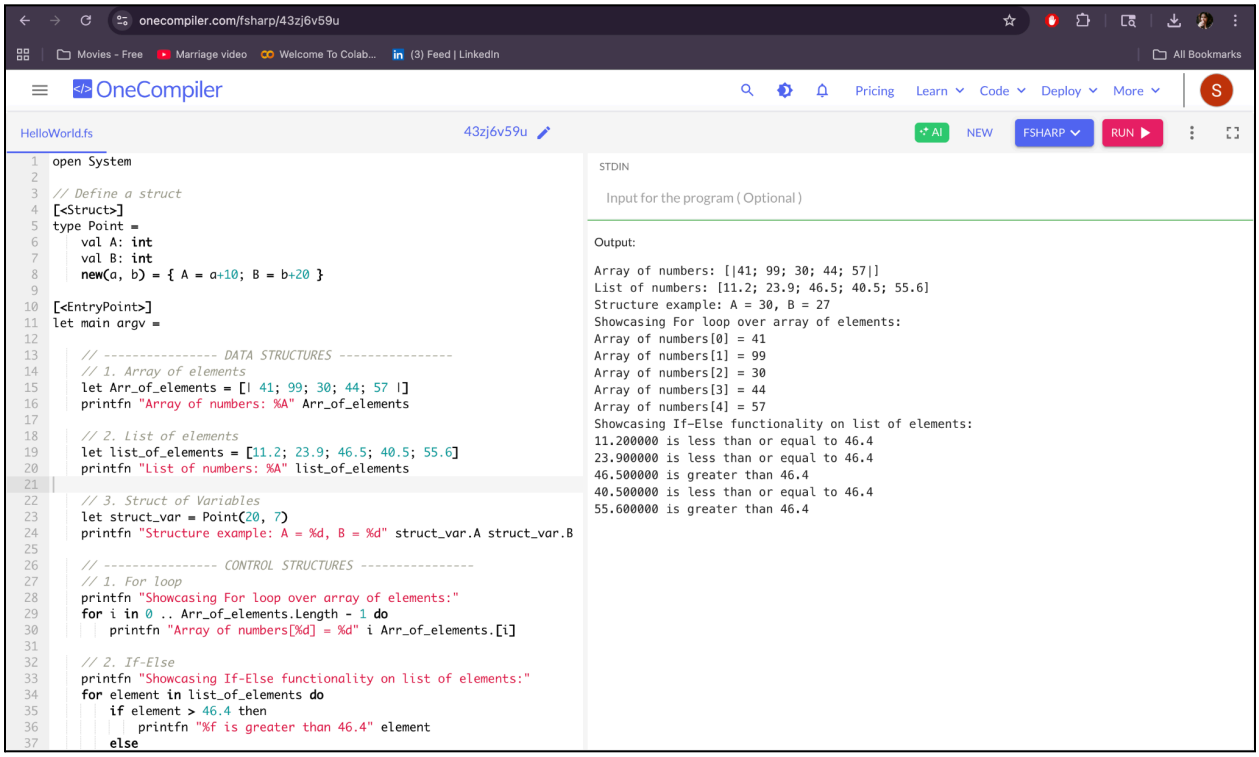


- 2 major data structures (array, list, struct) and 2 major control structure(example: for loop, while loop, if-else, lambda functions etc)

Code:

```
open System

// Define a struct
[<Struct>]
type Point =
    val A: int
    val B: int
    new(a, b) = { A = a+10; B = b+20 }

[<EntryPoint>]
let main argv =

    // ---------- DATA STRUCTURES ----------
    // 1. Array of elements
    let Arr_of_elements = [| 41; 99; 30; 44; 57 |]
    printfn "Array of numbers: %A" Arr_of_elements

    // 2. List of elements
    let list_of_elements = [11.2; 23.9; 46.5; 40.5; 55.6]
    printfn "List of numbers: %A" list_of_elements

    // 3. Struct of Variables
    let struct_var = Point(20, 7)
    printfn "Structure example: A = %d, B = %d" struct_var.A struct_var.B

    // ---------- CONTROL STRUCTURES ----------
    // 1. For loop
    printfn "Showcasing For loop over array of elements:"
    for i in 0 .. Arr_of_elements.Length - 1 do
        printfn "Array of numbers[%d] = %d" i Arr_of_elements.[i]

    // 2. If-Else
    printfn "Showcasing If-Else functionality on list of elements:"
    for element in list_of_elements do
        if element > 46.4 then
            printfn "%f is greater than 46.4" element
        else
            printfn "%f is less than or equal to 46.4" element
```

0 // successful execution



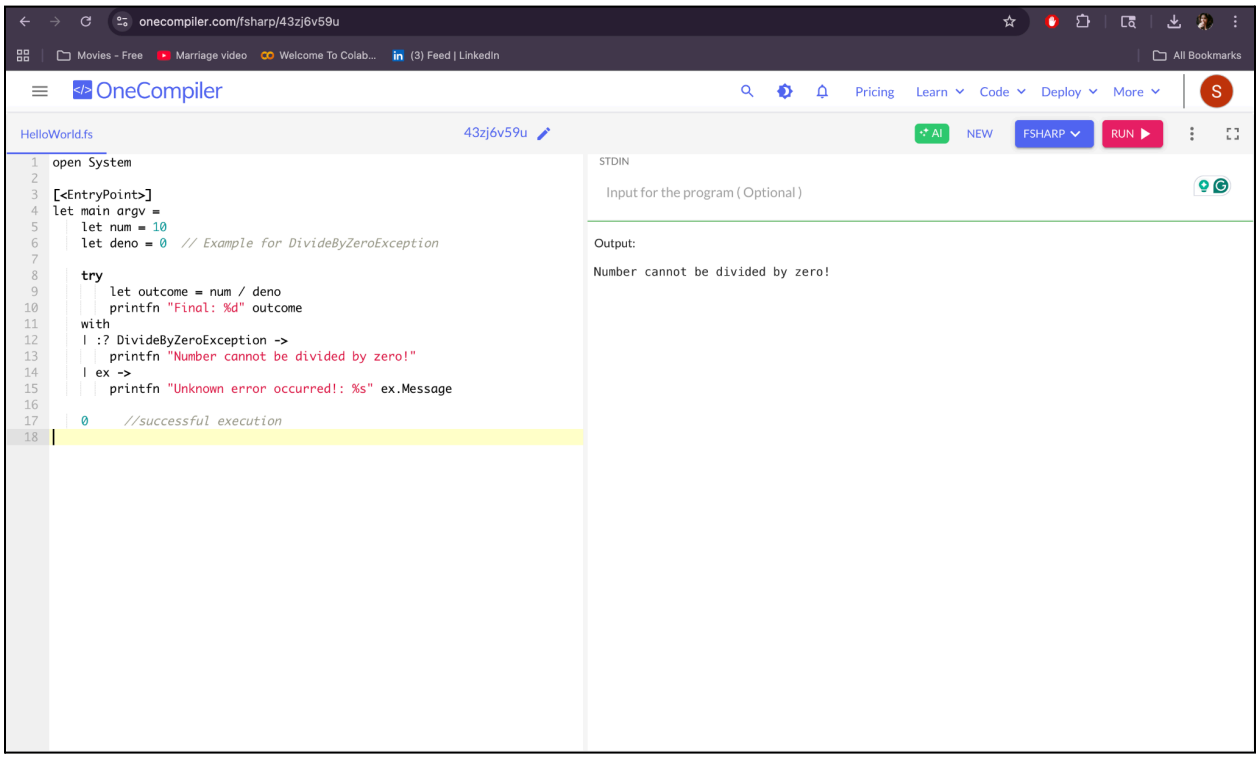
- showcasing exception handling or Concurrency.

Code:

```
open System

[<EntryPoint>]
let main argv =
    let num = 10
    let deno = 0  // Example for DivideByZeroException

    try
        let outcome = num / deno
        printfn "Final: %d" outcome
    with
    | :? DivideByZeroException ->
        printfn "Number cannot be divided by zero!"
    | ex ->
```

printfn "Unknown error occurred!: %s" ex.Message

0 //successful execution



Use Cases regarding Problem definition

- User Registration and Authentication

Actors: Client (User), System

- A new user signs up on the platform to participate in group therapy. They create an account, log in, and then get to their personal dashboard.
- The user goes to the registration page.
- They fill out the required details, like their username, password, email, and what role they're interested in.
- The system checks everything out and makes an account.
- Once the registration is all set, the user logs in to get started.
- If they ever forget their password, they can easily reset it.
- The system should also make sure email addresses are unique.

- Therapist Scheduling a Group Therapy Session

Actors: Therapist, System

- The therapist sets up a group therapy session, picks a date and time, and gets everyone in on the fun!
- The therapist logs in to the system.
- The therapist heads over to the "Schedule a Session" page.
- The therapist fills in all the important details, like what the session will be called, when it's happening, how long it'll last, and what everyone hopes to get out of it.
- The system checks everything to make sure it's all good.
- Once the session is all set, the system lets the members RSVP!

- Group Member Joining a Scheduled Therapy Session

Actors: Client (User), Therapist, System

- A user logs into their therapy session, which the therapist has set up.
- They then find their way to the "Upcoming Sessions" section.
- The system shows all the sessions they've been invited to.
- The client picks a session they'd like to join.

Proposed Method and its intuition

The intuition for using a functional programming based Fsharp language is that by reducing the hidden side effects and enforcing predictable transformation, the system becomes more reliable and easier to maintain compared to typical imperative or mixed-paradigm backend execution. This approach is beneficial over the traditional CRUD APIs because it ensures type safety, clear data flow and functional composition, that lessens the runtime errors and enhances code readability, especially valuable in sensitive domains like healthcare and group therapy management.

Core Algorithms in F# Group Therapy API

- Session Management Algorithm
    - We use records that can't be changed to keep track of therapy sessions.
    - We make sure that only the right steps happen, *like when a session starts, becomes active*, and then closes.
    - This way, we can't do things like add people to a session that's already closed.
- Participant Registration Algorithm

    - It checks that the participant data is correct using discriminated unions (Patient | Therapist).
    - It sends back either a success or an error using the Result type, so we don't have to worry about exceptions.
    - It makes sure that each participant in a session is unique.
- Message Exchange Algorithm
    - It uses async workflows (async {}) to manage messages that happen at the same time.
    - It uses queues or in-memory collections to mimic a group chat.
    - It makes sure messages are delivered in the right order and that you get a confirmation when they arrive.
- Data Integrity Algorithm
    - It uses the Option type to manage missing values in a safe way.
    - Pattern matching ensures that null-like situations are handled with precision.
    - By making the "absence of data" clear, it helps prevent runtime crashes.
- Error Handling Algorithm
    - It uses Result<'T,'Error> to pass errors along the pipeline.
    - It makes you think about how to put things together in a functional way (like binding and mapping) to handle errors nicely.
    - It keeps the code from getting too bogged down in exceptions, which makes it more robust.
- Concurrency Control Algorithm
    - Async workflows handle multiple people at once!
    - Tasks are set up to run without blocking, so you'll always be in the loop.
    - You can even use them to mimic having dozens of therapy clients working together.
- Persistence Algorithm
    - It takes records and turns them into JSON using System.Text.Json.
    - It makes sure everything follows the same rules with discriminated unions.
    - It gives you consistent session logs that you can use to look at things.

Experiments

We started by creating a simple REST API using dotnet new webapi - language F# to get a basic structure and test out some endpoints.

Next, we explored functional routing with frameworks like *Falco* or *Saturn*, which let me define routes using functions and pattern matching.

To handle errors more gracefully in API responses, we switched from exceptions to Option and Result types.

we also experimented with F#'s async {} workflows to manage concurrent requests effectively.

For modeling API inputs and outputs, we used discriminated unions and records to ensure everything is strongly typed.

Finally, we integrated with existing C#/.NET libraries within my F# API to blend different programming paradigms.

Description of Your Test Bed

Our test bed is a controlled environment where we've been testing and checking out our F# API for group therapy. Here's what it includes:

- **API Framework:** We've built it in F# using ASP.NET Core to show off the endpoints for therapy sessions, participants, and all the interactions.
- **Data Model:** We've created records and discriminated unions to represent *patients*, *sessions*, *users*, and *rsvps*.
- **Endpoints:** We've set up RESTful routes for creating sessions, getting participants to join, posting therapy notes, and checking out group progress.
- **Deployment Environment:** We've got a local test server running (using dotnet run).
- **Test Clients:** We've got scripts or tools (like Postman, express js, or F# scripts) to help us simulate multiple participants using the API.
- **Monitoring Tools:** We're using logging middleware, performance counters, and other tools to make sure everything is working as expected.

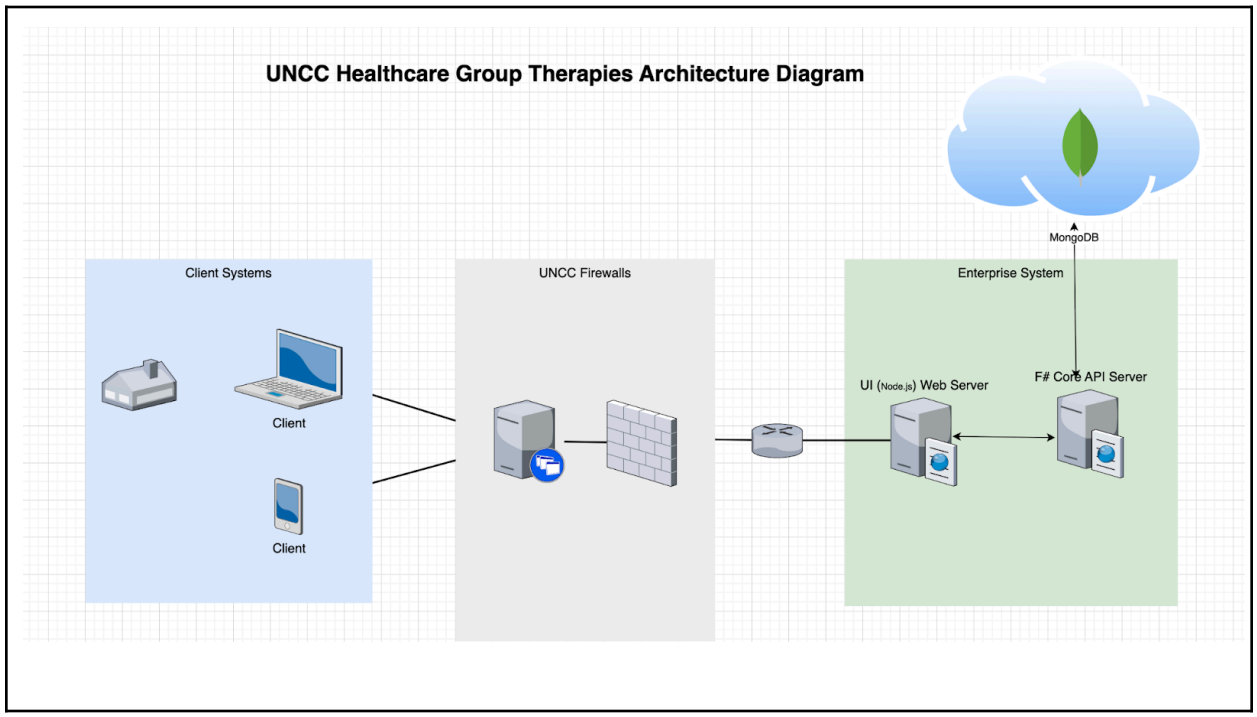
This test bed is like a virtual group therapy platform where lots of clients can interact with the API at the same time, just like they would in real life.

List of Questions Your Experiments Are Designed to Answer

- Scalability: How does the F# API manage multiple people working together during a therapy session?
- Data Integrity: Does using F#'s Option and Result types help prevent runtime errors compared to using exceptions?

- Performance: What are the latency and speed of the API when it's under a simulated group load?
- Concurrency: How well do F# async workflows handle multiple therapy interactions at the same time?
- Domain Modeling: Do discriminated unions and records make it easier and safer to define the rules for therapy session data?
- Interoperability: How easy is it to connect the F# API with other C#/.NET libraries for things like signing in, keeping track of logs, or saving data?
- Error Handling: Does using functional error handling (like pattern matching and results) make the system more robust than using imperative methods?
- User Experience Simulation: How does the API help with features like scheduling sessions, chatting with participants, and keeping therapist notes in a real-world setting?

Project Architecture:

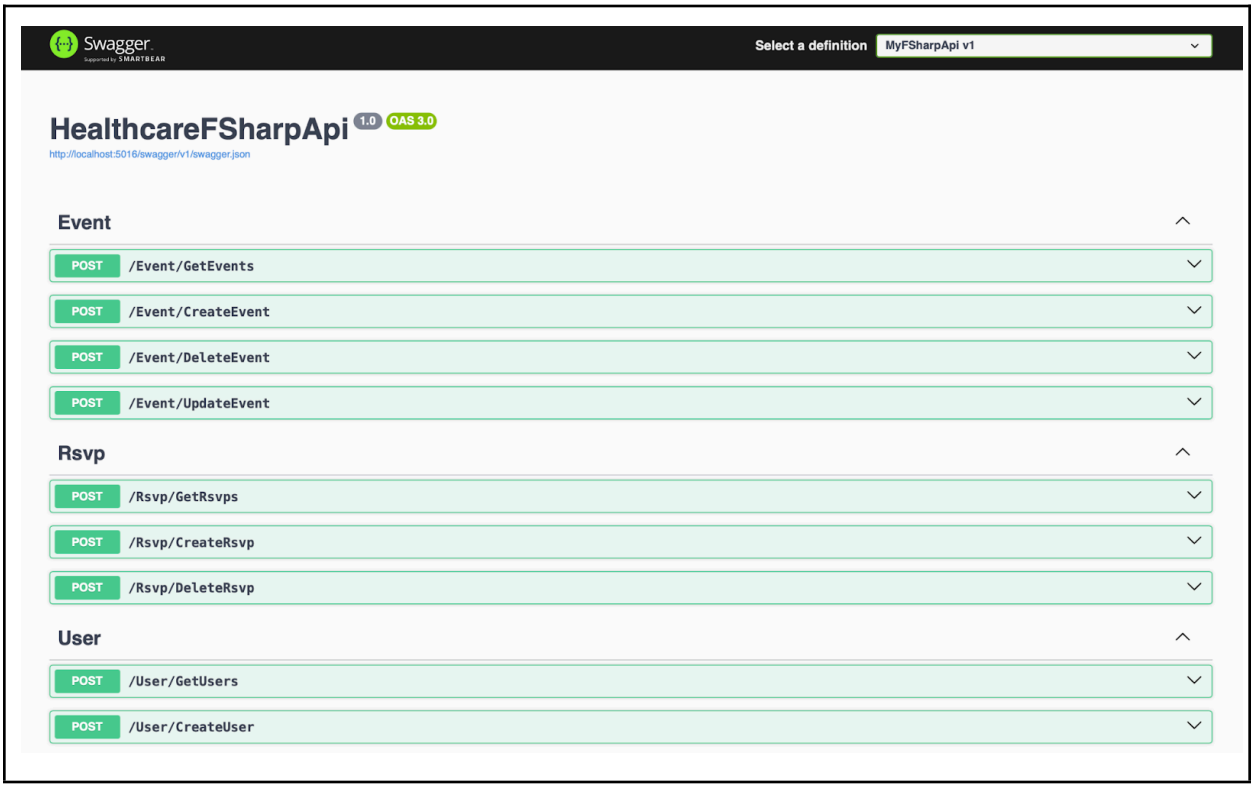


MongoDB Connection String:
'mongodb+srv://demo:demo123@cluster0.pcywg4i.mongodb.net/spl-project1?retryWrites=true&w=majority&appName=cluster0'

User Credentials:

User Email	Password
doraemon@uncc.com	asdf1234
nobita@uncc.com	asdf1234

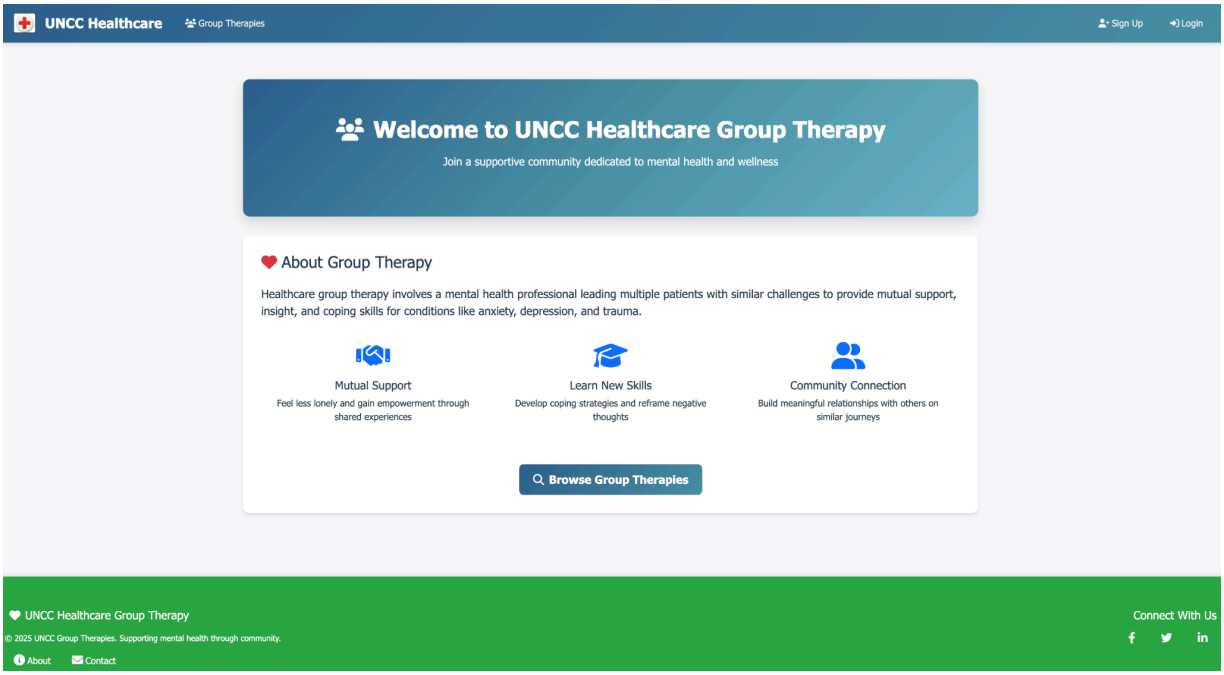
Core Business F# APIs / Services



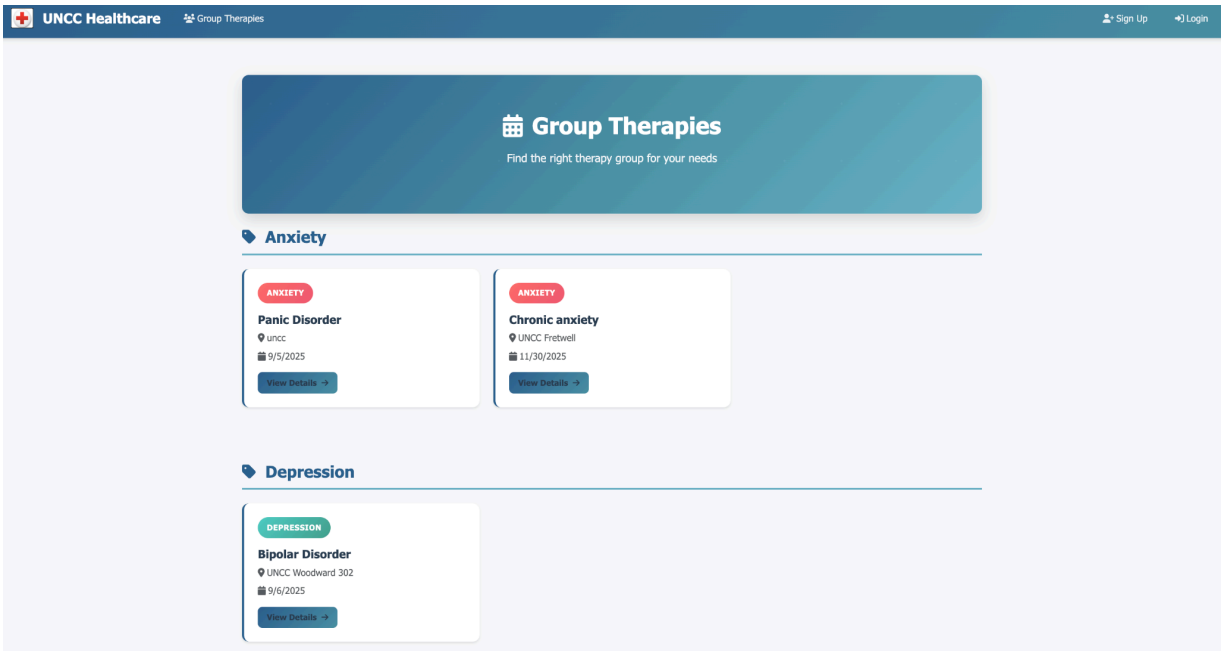
Details of Experiments/Observations:

Google Chrome Screen captures:

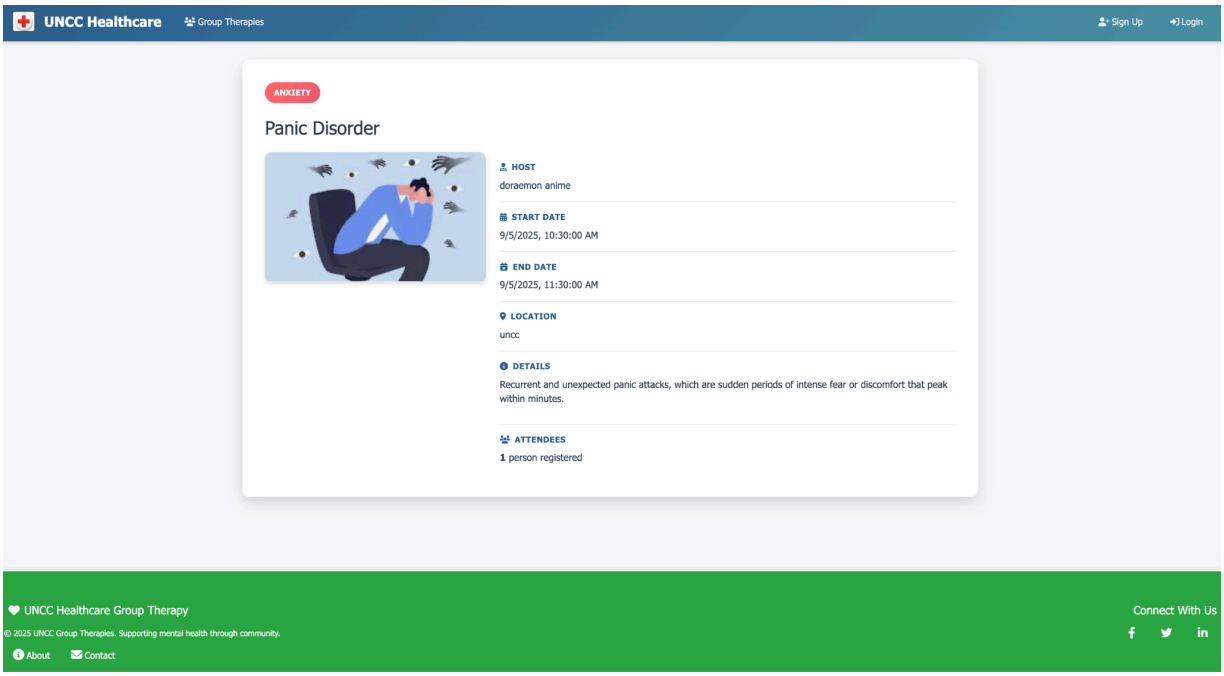
Logged out user app/main screen capture:



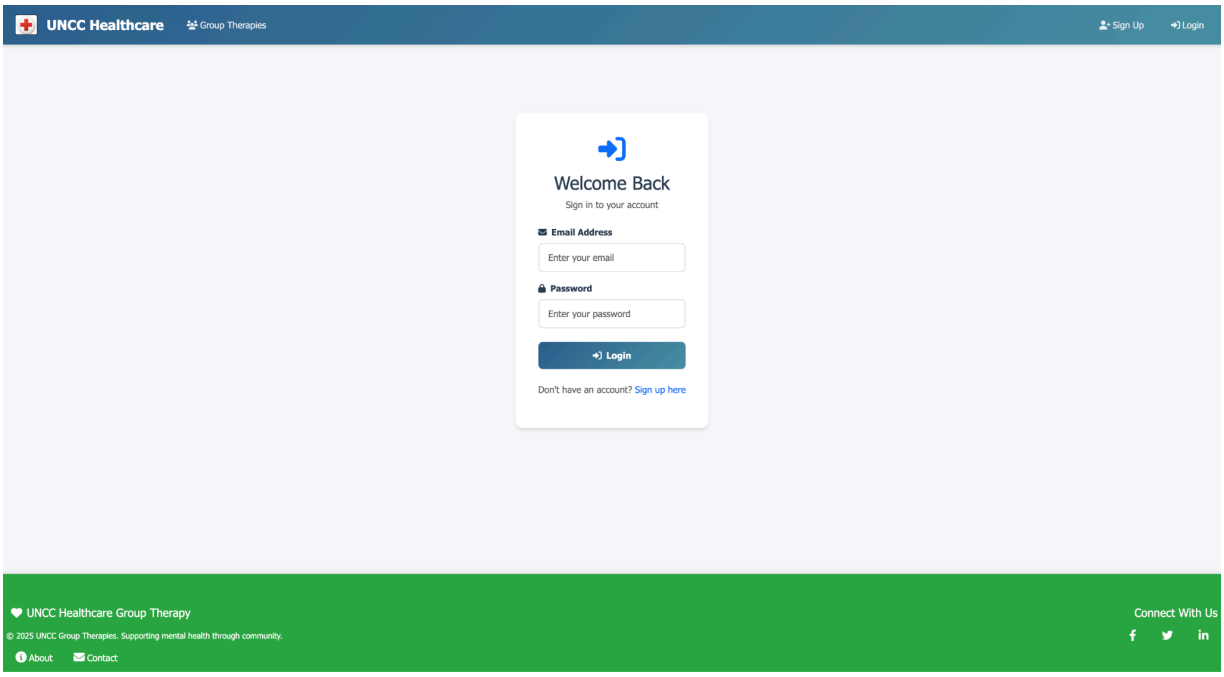
Logged out user events screen capture:



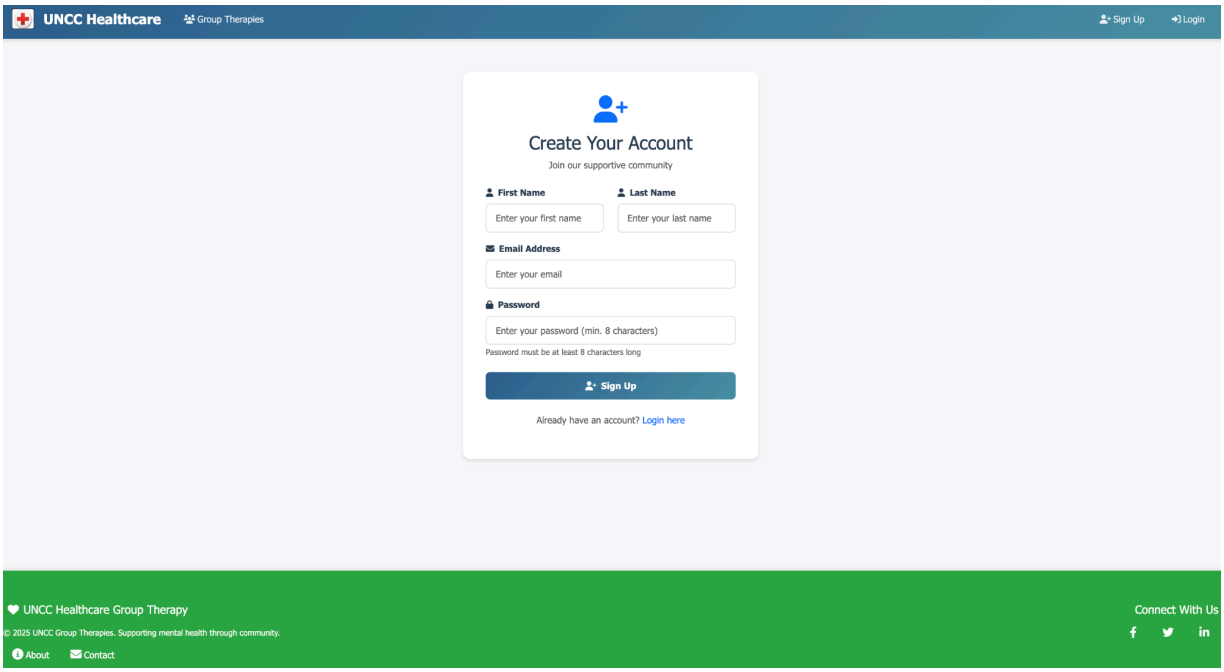
Logged out user view/show event screen capture:



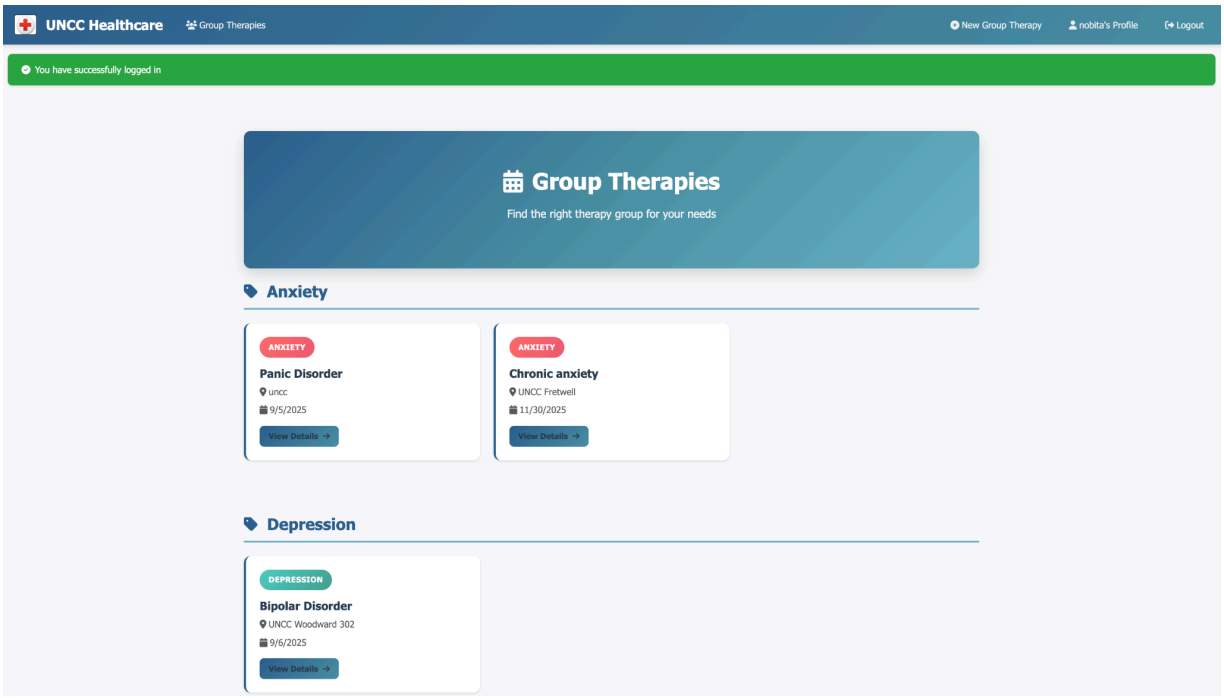
Log In user screen capture:



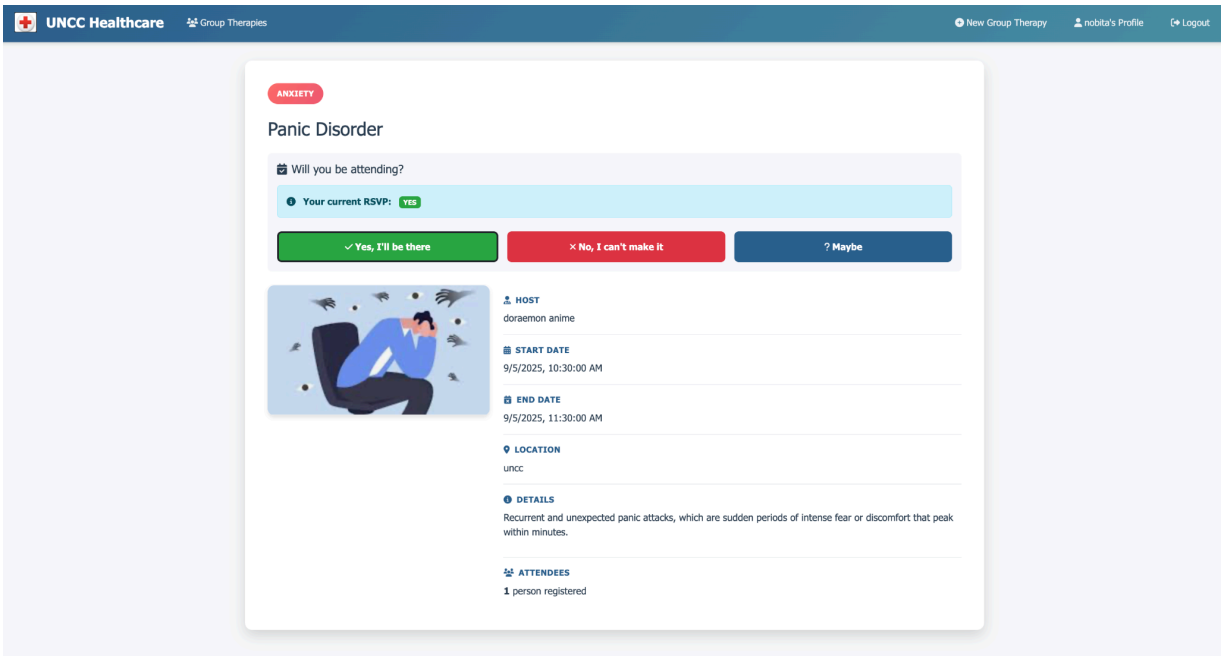
Sign Up user screen capture:



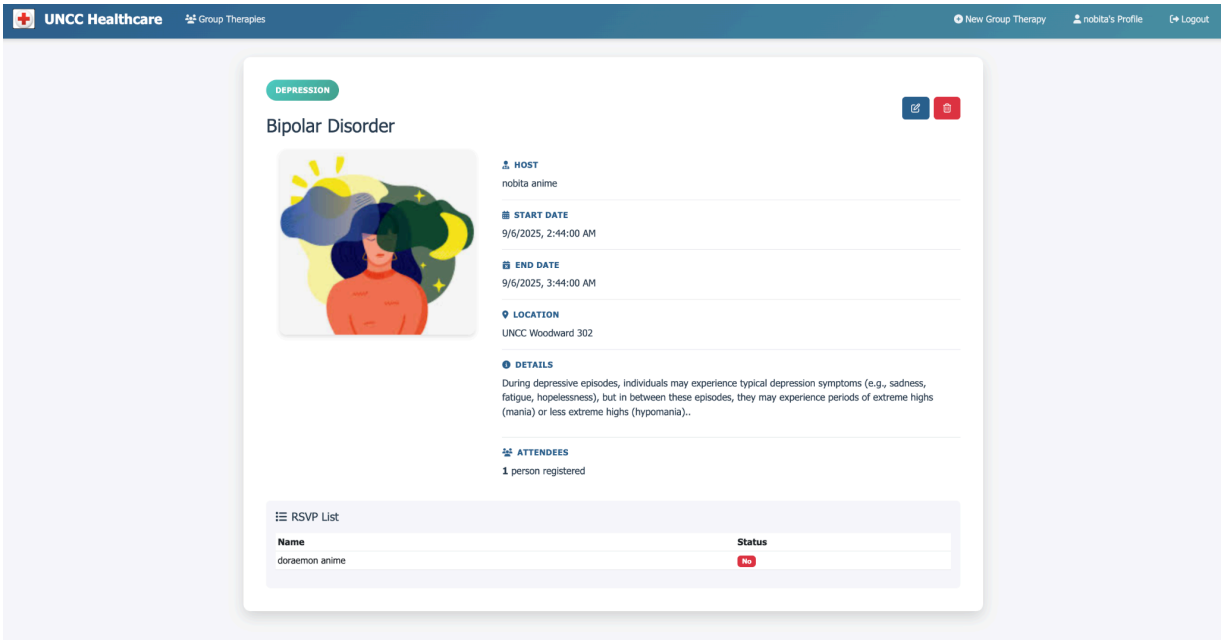
Logged In user (doraemon) screen capture:



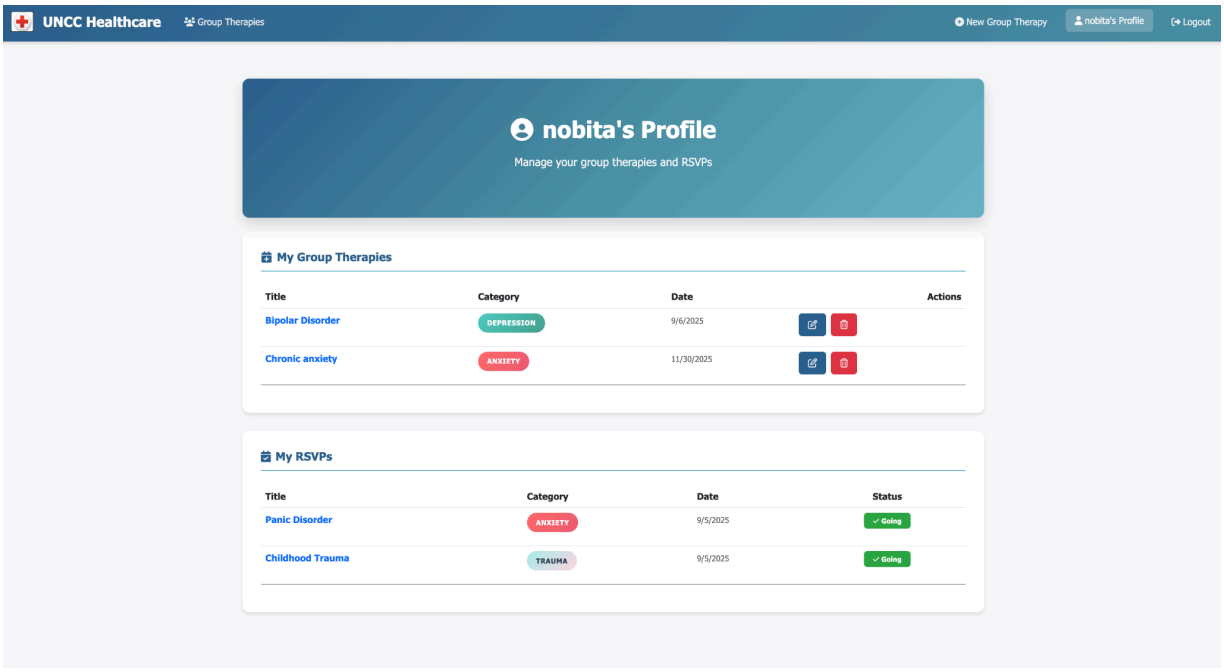
Logged In user (doraemon) selected event detail of other user (nobita) screen capture:



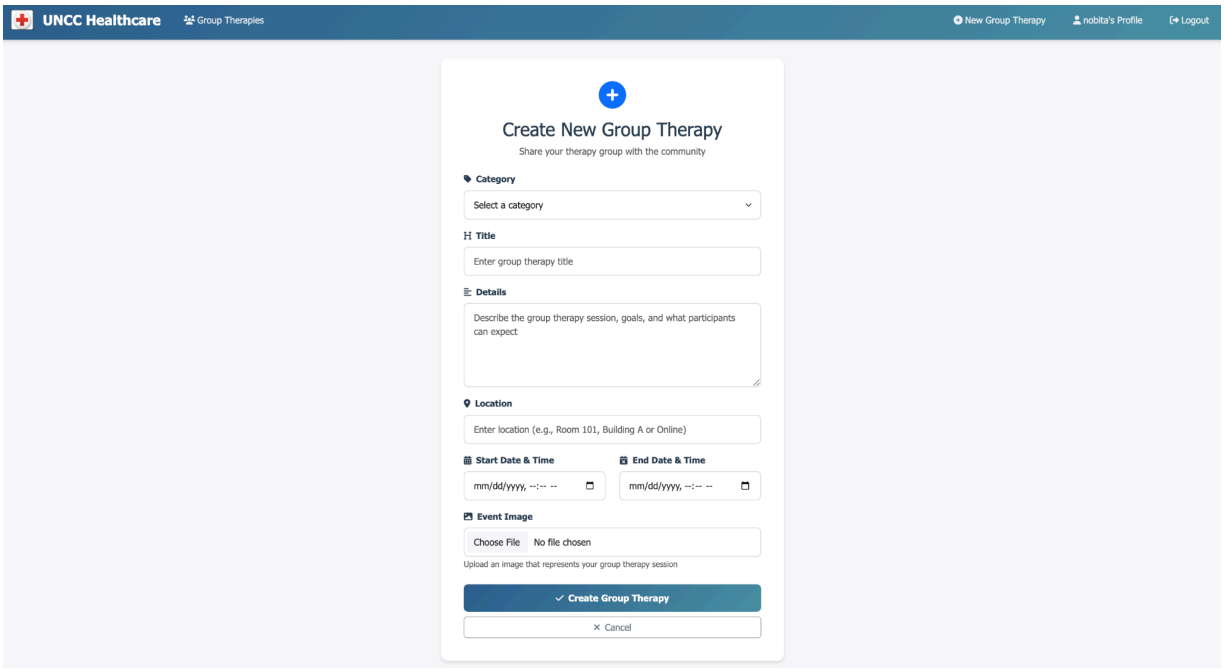
Logged In user (doraemon) selected his own event detail screen capture:



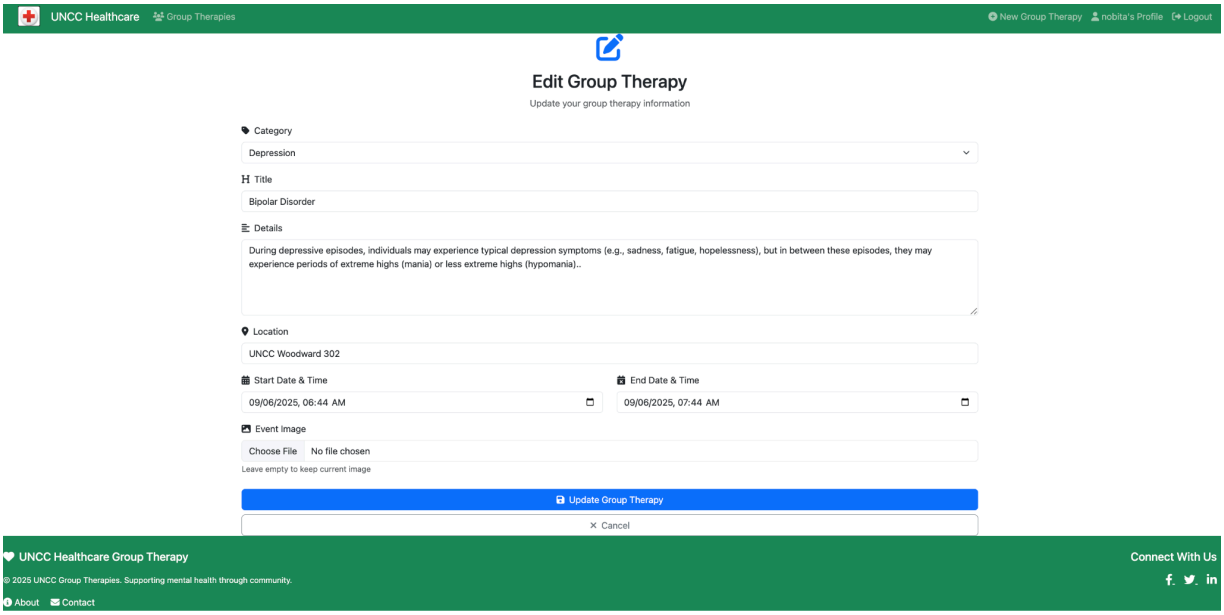
Logged In user (doraemon) profile screen capture:



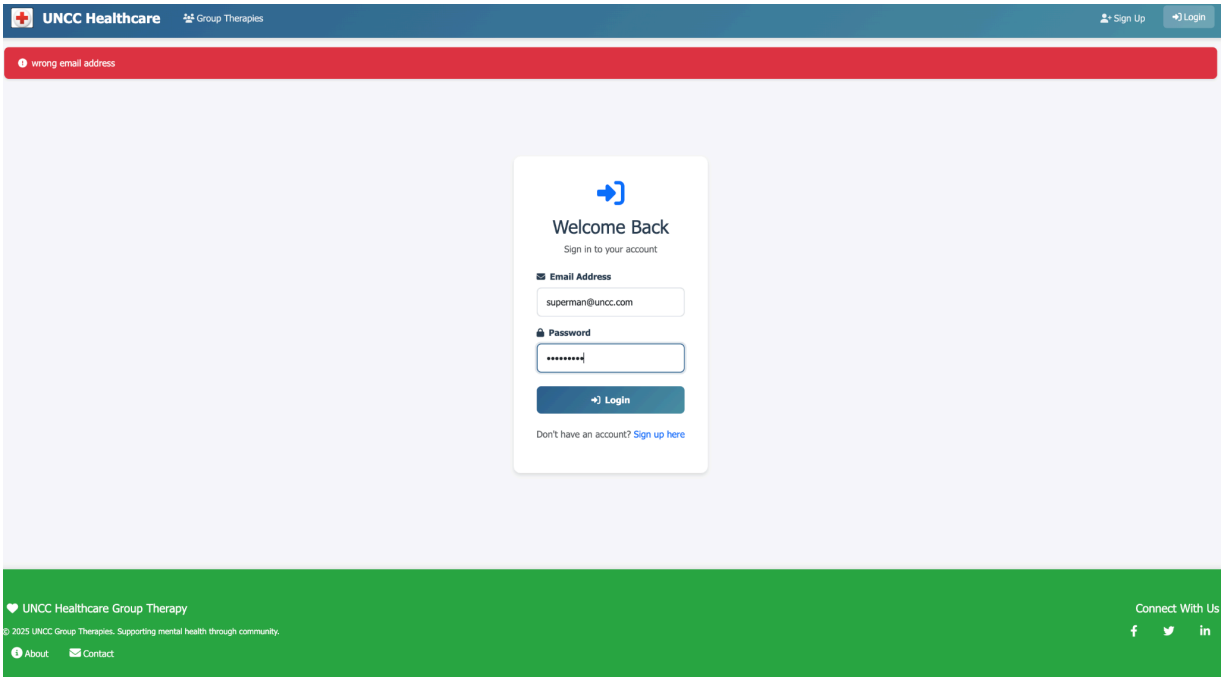
Logged In user (doraemon) new event screen capture:



Logged In user (doraemon) edit event detail screen capture:



Not Registered User tried to login



Observations

The project executes a functional F# backend for handling users, events and RSVPs, leveraging strongly typed records and data structures that are immutable to make sure of predictable behavior. It combines MongoDB for steady storage, with secure management of ObjectIDs and JSON serialization for API replies. Controllers like the EventController, RsvpController and the UserController give CRUD operations while showcasing F# features like pattern matching, list processing and record-based data modelling. Overall, it highlights the way F# can be leveraged effectively for a real-world, healthcare-focused group therapy management system.

Conclusion

The group therapy API test bed shows how F# functional programming ideas can help create dependable, adaptable, and expressive systems for working together. It uses things like immutable data structures, discriminated unions, and functional error handling to make sure session management and how participants talk to each other are spot-on. Plus, async workflows make sure that concurrency is managed nicely, which is like how people in a group usually interact at the same time.

In our experiments, we looked into how F#'s focus on functions helps with scalability, keeping data safe, making things run smoothly, and working together with other systems. We found that this design not only cuts down on runtime errors but also makes it easier to understand the agreements for complicated areas, like therapy sessions. The algorithms we created, from managing session lives to sending messages, show how using functional ideas can make tricky logic simpler and less likely to go wrong.

This project really shows that F# is a fantastic language for building APIs in important areas like group therapy. It's all about making sure things are strong, easy to understand, and can handle whatever comes its way. The test environment is a great starting point for what's next, like connecting with authentication systems, saving data, and putting everything up in the cloud. This will open up doors for even more cool stuff in digital therapy platforms.
