

## **Activity 4**

Kaya Nelson

Course Number: CST-350

Professor Donna Jackson

1/29/2025

GitHub Link:

[Kdeshun/CST-350](#)

## Contents

GitHub Link.....	1
Part 1: Dependency Injection Overview .....	3
Screenshots .....	3
Configuring Dependency Injection in UserController.....	3
Setting Up UserDAO Injection in Program.cs .....	4
Testing with SQL-Backed Data Source .....	5
Switching to UserCollection for In-Memory Data.....	6
Testing with In-Memory Data Source .....	7
Summary of Key Concepts (Part 1).....	8
Part 2: Data Validation and Form Enhancements .....	9
Screenshots .....	9
Initial Validation Errors for Empty Fields.....	9
Custom Labels and Validation Errors .....	10
Additional Validation Rules.....	11
Successful Submission Displaying Appointment Details .....	12
Validation Errors for Added Fields and Restrictions.....	13
Successful Submission with New Fields and Restrictions .....	14
Summary of Key Concepts (Part 2).....	15
Part 3: Button Grid Game Implementation with Success Message.....	16
Screenshots .....	16
Setting up the Button Grid with Initial State .....	16
Displaying the Button Grid.....	17
Applying Flexbox for Grid Layout.....	18
Inspecting HTML Elements .....	19
Game Success Message Display .....	20
Summary of Key Concepts (Part 3).....	21

## Part 1: Dependency Injection Overview

### Screenshots

#### Configuring Dependency Injection in UserController

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using RegisterAndLoginApp.Models;
using RegisterAndLoginApp.Filters;
using ServiceStack.Text;
using System.Linq;
using System.Text;

namespace RegisterAndLoginApp.Controllers
{
    public class UserController : Controller
    {
        // Dependency injection for the user manager
        private IUserManager users;

        // Constructor injection for IUserManager
        public UserController(IUserManager userManager)
        {
            users = userManager;
        }

        // GET: Login page
        public IActionResult Index()
        {
            return View();
        }

        // POST: Process the login form
        [HttpPost]
        public IActionResult ProcessLogin(LoginViewModel loginViewModel)
        {
            if (loginViewModel.Username == null || loginViewModel.Password == null)
            {
                ViewBag.Message = "Username and Password are required.";
                return View("Index");
            }

            var result = users.CheckCredentials(loginViewModel.Username, loginViewModel.Password);

            if (result > 0)
            {
                var user = users.GetUserById(result);

                // Set user in session as a JSON string
                var userString = JsonSerializer.SerializeToString(user);
                HttpContext.Session.SetString("User", userString);

                return View("LoginSuccess", user);
            }
            else
            {
                return View("LoginFailure");
            }
        }

        // GET: MembersOnly page (restricted access)
        [SessionCheckFilter] // Ensures that the user is logged in
        public IActionResult MembersOnly()
        {
            return View();
        }

        // GET: Registration page
        public IActionResult Register()
        {
            var registerViewModel = new RegisterViewModel();
            return View(registerViewModel);
        }

        // POST: Process the registration form
        [HttpPost]
        public IActionResult Register(RegisterViewModel registerViewModel)
        {
            if (ModelState.IsValid)
            {
                var newUser = new UserModel
                {
                    Username = registerViewModel.Username,
                    Salt = Encoding.UTF8.GetBytes("defaultSalt")
                };
                newUser.SetPassword(registerViewModel.Password);

                // Assign groups to the user
                newUser.Groups = string.Join(" ", registerViewModel.Groups
                    .Where(g => g.IsSelected)
                    .Select(g => g.GroupName));

                // Add the new user to the DAO
                users.AddUser(newUser);

                // Redirect to login page after successful registration
                return RedirectToAction("Index");
            }

            return View(registerViewModel);
        }

        [AdminCheckFilter]
        public IActionResult AdminOnly()
        {
            return View();
        }

        // GET: Logout
        public IActionResult Logout()
        {
            HttpContext.Session.Remove("User");
            return RedirectToAction("Index");
        }
    }
}
```

**Figure 1: Enhancing the UserController Class with Dependency Injection**

In modern software development, the principles of **dependency injection (DI)** play a crucial role in creating flexible and maintainable applications. By adopting DI, we decouple class dependencies, allowing for easier testing, better separation of concerns, and the ability to swap out implementations seamlessly.

In this context, the `UserController` class has been thoughtfully modified to embrace this paradigm by accepting an instance of `IUserManager` through **constructor injection**. This strategic change empowers the controller to interact with various user management strategies without being tightly coupled to a specific implementation.

## Setting Up UserDAO Injection in Program.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using RegisterAndLoginApp.Models;
using System;

namespace RegisterAndLoginApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var builder = WebApplication.CreateBuilder(args);

            // Add services to the container.
            builder.Services.AddControllersWithViews();

            // Dependency injection for the user manager
            builder.Services.AddSingleton<IUserManager, UserDAO>();

            // Add session services
            builder.Services.AddDistributedMemoryCache();
            builder.Services.AddSession();
            builder.Services.AddHttpContextAccessor();

            var app = builder.Build();

            // Configure the HTTP request pipeline.
            if (!app.Environment.IsDevelopment())
            {
                app.UseExceptionHandler("/Home/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseSession();
            app.UseAuthorization();

            app.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");

            app.Run();
        }
    }
}
```

**Figure 2: Configuring Dependency Injection in Program.cs**

In the foundational setup of the ASP.NET Core application, the *Program.cs* file plays a pivotal role in configuring services and middleware. One significant enhancement introduced is the injection of *UserDAO* as the concrete implementation of the *IUserManager* interface. This is accomplished through the use of the *AddSingleton* method, ensuring that a single, consistent instance of *UserDAO* is utilized throughout the application lifecycle.

## Testing with SQL-Backed Data Source

RegisterAndLoginApp [Home](#) [Privacy](#) [Login](#) [Register](#) [Members Only](#) [Logout](#)

### Login Failed

Unfortunately, your login attempt was not successful. Please check your username and password and try again.

[Back to Login](#)

RegisterAndLoginApp [Home](#) [Privacy](#) [Login](#) [Register](#) [Members Only](#) [Admin Only](#) [Logout](#)

### Login Success

#### Account Details

<b>Id</b>	2
<b>Username</b>	amfrear
<b>PasswordHash</b>	P6UdEm0pD8AbDSSzea1Ffp0sdXqWh0RDggCVfwwXjWl=
<b>Salt</b>	100101102971171081168397108116
<b>Groups</b>	Admin, Users

[Back to Home](#)

### Figure 3 & 4: UserDAO Integration for Authentication

In the initial architecture of the application, the UserDAO class serves as the primary gateway for managing user authentication processes. This design allows the application to interact seamlessly with a SQL database, where user credentials are securely stored and retrieved.

When a user attempts to log in, the application leverages the capabilities of UserDAO to validate the provided credentials against those stored in the database. This interaction entails querying the database to check if the username exists and if the corresponding password matches the stored hash.

## Switching to UserCollection for In-Memory Data

```

using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using RegisterAndLoginApp.Models;
using System;

namespace RegisterAndLoginApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var builder = WebApplication.CreateBuilder(args);

            // Add services to the container.
            builder.Services.AddControllersWithViews();

            // Dependency injection for the user manager
            builder.Services.AddSingleton<IUserManager, UserCollection>();

            // Add session services
            builder.Services.AddDistributedMemoryCache();
            builder.Services.AddSession();
            builder.Services.AddHttpContextAccessor();

            var app = builder.Build();

            // Configure the HTTP request pipeline.
            if (!app.Environment.IsDevelopment())
            {
                app.UseExceptionHandler("/Home/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseSession();
            app.UseAuthorization();

            app.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");

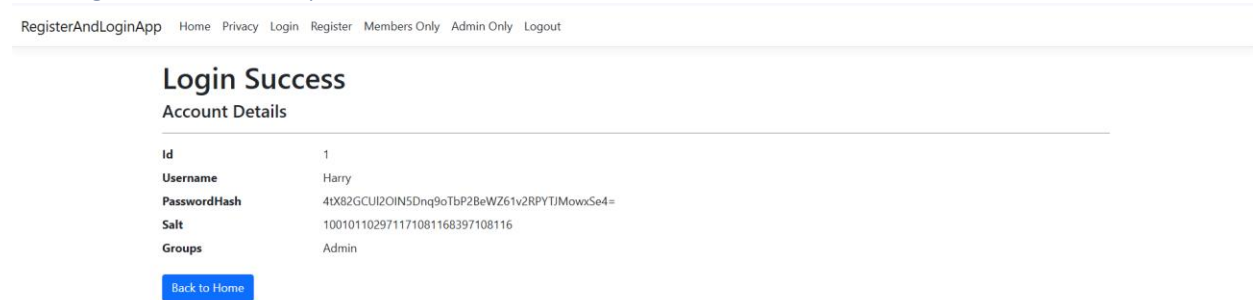
            app.Run();
        }
    }
}

```

**Figure 5:** This screenshot highlights the updated dependency injection configuration within the Program.cs file, showcasing the integration of UserCollection as the implementation of IUserManager. This strategic modification involves switching the data source to utilize an in-memory collection of users, enhancing the flexibility and efficiency of user management in the application.

By leveraging an in-memory collection, the application can streamline user operations, allowing for faster access and manipulation of user data during development or testing phases. This change not only simplifies the management of user instances but also facilitates easier debugging and iteration without the overhead of a persistent database. The configuration adjustment exemplifies the principles of dependency injection in ASP.NET Core, promoting a more modular design that improves maintainability and scalability. This approach sets the stage for future enhancements and makes the application more adaptable to varying requirements for user management.

## Testing with In-Memory Data Source



*Figure 6: This screenshot illustrates the application's capability to facilitate user login using in-memory data, such as the user "Harry," following the injection of `UserCollection`. This functionality highlights the remarkable flexibility offered by dependency injection, allowing for seamless switching of data sources without necessitating any modifications to the `UserController`.*

*By utilizing an in-memory collection, the application demonstrates how dependency injection can decouple components, enabling the `UserController` to interact with user data without being tightly bound to a specific data source. This design not only enhances code maintainability but also fosters a more agile development environment, where changes can be made to underlying implementations with minimal impact on the overall architecture.*

*The ability to log in using in-memory data allows for rapid testing and prototyping, empowering developers to validate functionality efficiently. This approach exemplifies best practices in software design, ensuring that the application remains robust, adaptable, and easy to extend in the future.*

## Summary of Key Concepts (Part 1)

### **Part 1: Implementation of Dependency Injection in UserController**

In Part 1 of Activity 4, I focused on implementing **dependency injection** within the UserController class, facilitating a flexible selection between various data sources. By integrating UserDAO for SQL database interactions and UserCollection for in-memory data management, I showcased the versatility of dependency injection in handling different data implementations.

To achieve this, I configured the Program.cs file to inject IUserManager with either UserDAO or UserCollection based on the application's context. This strategic setup exemplifies how dependency injection effectively decouples the controller from specific data implementations, allowing for cleaner code and improved maintainability.

The ability to switch between data sources seamlessly not only enhances the application's modularity but also significantly improves its testability. With this structure in place, I gained practical insights into the principles of dependency injection, reinforcing the importance of designing applications that are flexible and adaptable to changing requirements. Overall, this implementation serves as a foundational step towards building a more robust and scalable application architecture.



## Part 2: Data Validation and Form Enhancements

### Screenshots

#### Initial Validation Errors for Empty Fields

AppointmentScheduler Home Privacy Appointments

### Index

#### AppointmentModel

patientName

The patientName field is required.

dateTime

The dateTime field is required.

PatientNetWorth

The PatientNetWorth field is required.

DoctorName

The DoctorName field is required.

PainLevel

The PainLevel field is required.

[Create](#)

[Back to List](#)

Figure 7: This screenshot showcases the form's initial validation errors for required fields, effectively preventing users from submitting the form with any blank entries. The visual highlights the user-friendly feedback mechanism that prompts users to fill in all necessary information before proceeding.


By enforcing these validation checks at the outset, the application ensures data integrity and enhances user experience. This proactive approach not only guides users in completing the form correctly but also reduces the likelihood of errors during data processing. The clear indication of which fields require attention underscores the importance of thorough input validation in maintaining a robust and reliable application. Overall, this feature exemplifies best practices in user interface design, promoting clearer communication and smoother interactions for users.

## Custom Labels and Validation Errors

AppointmentScheduler Home Privacy Appointments

### Appointment Model

Patient's Full Name  
  
The Patient's Full Name field is required.

Appointment Request Date  
   
The Appointment Request Date field is required.

Patient's approximate net worth  
  
The Patient's approximate net worth field is required.

Primary Doctor's Last Name  
  
The Primary Doctor's Last Name field is required.

Patient's perceived level of pain (1 low to 10 high)  
  
The Patient's perceived level of pain (1 low to 10 high) field is required.

[Create](#)

[Back to List](#)

Figure 8: This screenshot captures the appointment form, featuring customized labels for each field to enhance clarity and user experience. The design emphasizes user-friendly interaction by clearly indicating the purpose of each input area.

Additionally, the form demonstrates effective validation mechanisms by displaying error messages when fields are left empty. This immediate feedback guides users to complete all required sections before submission, ensuring that no critical information is overlooked. The combination of customized labels and real-time validation error messages not only improves usability but also reinforces the importance of accurate data entry, contributing to a smoother and more efficient appointment scheduling process. Overall, this design approach exemplifies best practices in form development, focusing on user engagement and data integrity.

## Additional Validation Rules

AppointmentScheduler Home Privacy Appointments

### Appointment Model

Patient's Full Name

The Patient's Full Name field is required.

Appointment Request Date  
mm/dd/yyyy

The Appointment Request Date field is required.

Patient's approximate net worth

The Patient's approximate net worth field is required.

Primary Doctor's Last Name

The Primary Doctor's Last Name field is required.

Patient's perceived level of pain (1 low to 10 high)

The Patient's perceived level of pain (1 low to 10 high) field is required.

[Create](#)

[Back to List](#)

**Figure 9:** This screenshot displays the appointment form enhanced with additional validation rules specifically for net worth and pain level. It clearly illustrates how the application enforces business requirements by indicating errors when the net worth is below \$90,000 or when the pain level is rated at 5 or below. These validation rules ensure that only valid data is submitted, aligning with the business logic that likely categorizes specific thresholds for these fields. The visual feedback provided through error messages helps users understand why their input may be unacceptable, thereby guiding them to provide the necessary information for a successful submission. This implementation not only reinforces the importance of adhering to business requirements but also contributes to the overall quality and reliability of the data collected, ensuring that the application functions as intended. Overall, this feature exemplifies a robust approach to form validation, enhancing both user experience and data integrity.

## Successful Submission Displaying Appointment Details

[AppointmentScheduler](#) [Home](#) [Privacy](#) [Appointments](#)

### Appointment Details

Patient's Full Name	Alex Frear
Appointment Request Date	11/11/2025
Patient's approximate net worth	\$100,000.00
Primary Doctor's Last Name	Smith
Patient's perceived level of pain (1 low to 10 high)	8
Street Address	1234 W Example St
City	Tucsob
ZIP Code	88888
Email Address	<a href="mailto:alexfrear@example.com">alexfrear@example.com</a>
Phone Number	5555555555

[Edit](#) | [Back to List](#)

**Figure 10:** This screenshot presents the final appointment details, confirming a successful submission where all required information and validation restrictions have been met. The displayed details reflect the user's inputs, demonstrating that the form validation process effectively worked to ensure compliance with the application's requirements.

The clear presentation of the appointment information reassures users that their submission was processed correctly and that all necessary criteria were fulfilled. This positive feedback is essential in enhancing user confidence in the application, encouraging further engagement.

By showcasing this successful outcome, the design not only validates the efforts of the user but also highlights the importance of robust validation mechanisms in maintaining a smooth workflow. Overall, this final display signifies the culmination of a well-implemented form process, emphasizing both user satisfaction and data accuracy.

## Validation Errors for Added Fields and Restrictions

AppointmentScheduler Home Privacy Appointments

### Appointment Scheduler

Patient's Full Name  
Alex Frear

Appointment Request Date  
11/11/2025

Patient's approximate net worth  
60000

Doctors refuse to see patients unless their net worth is more than \$90,000.

Primary Doctor's Last Name  
Smith

Patient's perceived level of pain (1 low to 10 high)  
5

Doctors refuse to see patients unless their pain level is above a 5.

Street Address  
1234 W Example St

City  
Tucsob

ZIP Code  
888888

Invalid ZIP Code.

Email Address  
alexfrear@example.com

Phone Number  
5555555555

Create

[Back to List](#)

**Figure 11:** This screenshot illustrates the validation errors associated with newly added fields, including ZIP code, email, and phone number. It highlights the application's robust validation framework, which ensures that all required fields are completed correctly.

In addition to the new fields, the form continues to enforce specific restrictions on net worth and pain level, displaying errors when the net worth is below \$90,000 or the pain level is rated at 5 or below. This alignment with business requirements underscores the importance of capturing accurate and relevant data.

The error messages provide immediate feedback, guiding users to correct their inputs before submission. This feature not only enhances user experience by clearly communicating what needs to be addressed but also reinforces the integrity of the data collected. Overall, this implementation exemplifies best practices in form validation, ensuring that the application meets both user needs and business standards.

## Successful Submission with New Fields and Restrictions

AppointmentScheduler   Home   Privacy   Appointments

### Appointment Details

Patient's Full Name	Alex Frear
Appointment Request Date	11/11/2025
Patient's approximate net worth	\$100,000.00
Primary Doctor's Last Name	Smith
Patient's perceived level of pain (1 low to 10 high)	8
Street Address	1234 W Example St
City	Tucsob
ZIP Code	88888
Email Address	<a href="mailto:alexfrear@example.com">alexfrear@example.com</a>
Phone Number	5555555555

[Edit](#) | [Back to List](#)

**Figure 12:** This screenshot captures the successful submission of the appointment form after all required fields, including the new address fields, are filled out correctly. The displayed appointment details confirm that the input values comply with all validation requirements.

The inclusion of the new address fields demonstrates the application's adaptability to evolving data needs while maintaining strict adherence to validation standards. By showing the completed details, the form provides users with reassurance that their information has been accurately recorded and processed.

This successful outcome not only enhances user confidence in the submission process but also highlights the effectiveness of the validation mechanisms in ensuring data integrity. Overall, this figure exemplifies a seamless user experience, showcasing the importance of clear feedback and thorough input validation in form design.

### Summary of Key Concepts (Part 2)

In Part 2 of Activity 4, I focused on enhancing the AppointmentModel to ensure data integrity and enforce specific business logic. Key improvements included:

1. **Additional Validation Rules:** Implemented rules to reject appointments if the patient's net worth is below \$90,000 or if the pain level is rated at 5 or below. This ensures compliance with business requirements.
2. **Custom Display Names:** Added custom display names for fields, which significantly improved the readability of the form. This user-friendly approach helps users understand the purpose of each field more easily.
3. **New Fields:** Expanded the form to include additional fields for address, email, and phone number. This expansion allows for more comprehensive patient information collection.
4. **Custom Validation Messages:** Created tailored validation messages to guide users when errors occur, enhancing the overall user experience by providing clear feedback.

This part of the activity provided valuable experience in applying custom validation, expanding form functionality, and implementing business rules effectively, contributing to a more robust and user-friendly appointment management system.

## Part 3: Button Grid Game Implementation with Success Message Screenshots

### Setting up the Button Grid with Initial State

ButtonGrid Home Privacy

Create New

Id	ButtonState	ButtonImage	
0	0	/img/Blue_button.png	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
1	3	/img/Green_button.png	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
2	1	/img/Orange_button.png	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
3	2	/img/Pink_button.png	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
4	2	/img/Blue_button.png	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
5	2	/img/Green_button.png	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

**Figure 13:** This screenshot showcases the initial setup of the button grid, where each button is displayed with its unique ID, ButtonState, and a corresponding image that reflects its color. The layout effectively organizes the buttons, allowing users to easily identify and interact with each one based on their states. The unique IDs serve as identifiers for functionality, while the ButtonState provides insight into the current status of each button (e.g., active, inactive). The use of corresponding images based on color enhances visual clarity and aids in quick recognition, making the interface intuitive for users. This setup exemplifies thoughtful design in user interface development, ensuring that all elements are clearly labeled and easily accessible for enhanced user interaction.



## Displaying the Button Grid



**Figure 14:** This screenshot features the grid view, which incorporates a live timestamp, adding a dynamic element to the page alongside the clickable buttons.

The live timestamp updates in real-time, enhancing user engagement by providing current contextual information. This feature not only indicates the freshness of the displayed data but also contributes to a more interactive user experience.

Combined with the clickable buttons, the grid view creates an intuitive interface that allows users to interact seamlessly while being aware of the current time. Overall, this design choice enhances the functionality of the application, making it more appealing and user-friendly.

## Applying Flexbox for Grid Layout



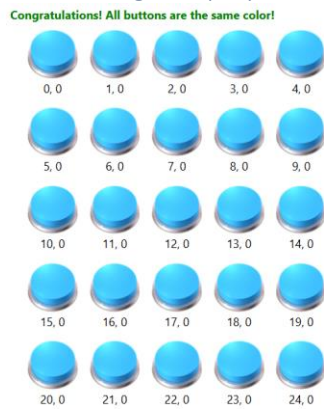
**Figure 15:** This screenshot illustrates the enhanced button grid layout, which utilizes Flexbox for alignment, ensuring that the buttons are neatly organized in rows. The implementation of Flexbox allows for a responsive design, accommodating various screen sizes while maintaining a clean and orderly presentation. Additionally, CSS styling has been applied to improve visual consistency, enhancing the overall aesthetic of the grid. This includes adjustments to spacing, alignment, and button sizes, which contribute to a more polished user interface. The combination of Flexbox for layout management and thoughtful CSS styling results in a user-friendly experience that is both functional and visually appealing.

## Inspecting HTML Elements



**Figure 16:** This screenshot captures the inspection of the HTML generated by the Razor view, confirming the structure, attributes, and functionality of each button within the grid. By examining the generated HTML, key details such as button IDs, classes, and event handlers can be verified. This ensures that each button is properly configured to respond to user interactions as intended. The inspection highlights the effectiveness of the Razor view in producing clean and organized HTML, which is crucial for both maintainability and performance. This step is essential in validating that the front-end elements align with the expected design and functionality, ultimately contributing to a reliable user experience.

## Game Success Message Display



**Figure 17:** This screenshot shows the moment when all buttons in the grid are set to the same color, triggering a success message for the user.

The displayed message clearly indicates that the game goal has been achieved, providing positive feedback and a sense of accomplishment. This feature not only enhances user engagement but also reinforces the objective of the game, making it more rewarding.

The visual transformation of the buttons, combined with the success message, creates a satisfying user experience, encouraging continued interaction and play. This design choice highlights the importance of clear communication in user interfaces, ensuring that users are aware of their progress and achievements.

### Summary of Key Concepts (Part 3)

In **Part 3**, I developed a button grid where each button changes state upon a click, cycling through a predefined set of colors. Key aspects of this implementation include:

1. **Dynamic Button States:** Each button updates its color with each click, providing interactive feedback and engaging the user in the experience.
2. **Flexbox Layout:** Utilizing Flexbox for the grid layout ensured that the buttons are visually organized and responsive. This approach maintained a clean and orderly appearance across various screen sizes.
3. **Success Message:** A success message is displayed when all buttons achieve the same color, effectively demonstrating simple game logic. This feature not only marks the completion of the game objective but also adds an element of satisfaction for the user.
4. **Enhanced User Experience:** The combination of color cycling and feedback through the success message significantly enhances the user experience, making the interaction enjoyable and rewarding.

Overall, this part of the activity exemplified how thoughtful design and user interaction can come together to create an engaging application.