CST-350 Activity 4 Dependency Injection, Data Validation, and ButtonGrid

Alex M. Frear

College of Science, Engineering, and Technology, Grand Canyon University

Course Number: CST-350

Professor Brandon Bass

11/17/2024

GitHub Link:

https://github.com/amfrear/cst350/tree/main/Activity_4

Contents

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");
        }
    }
}
```

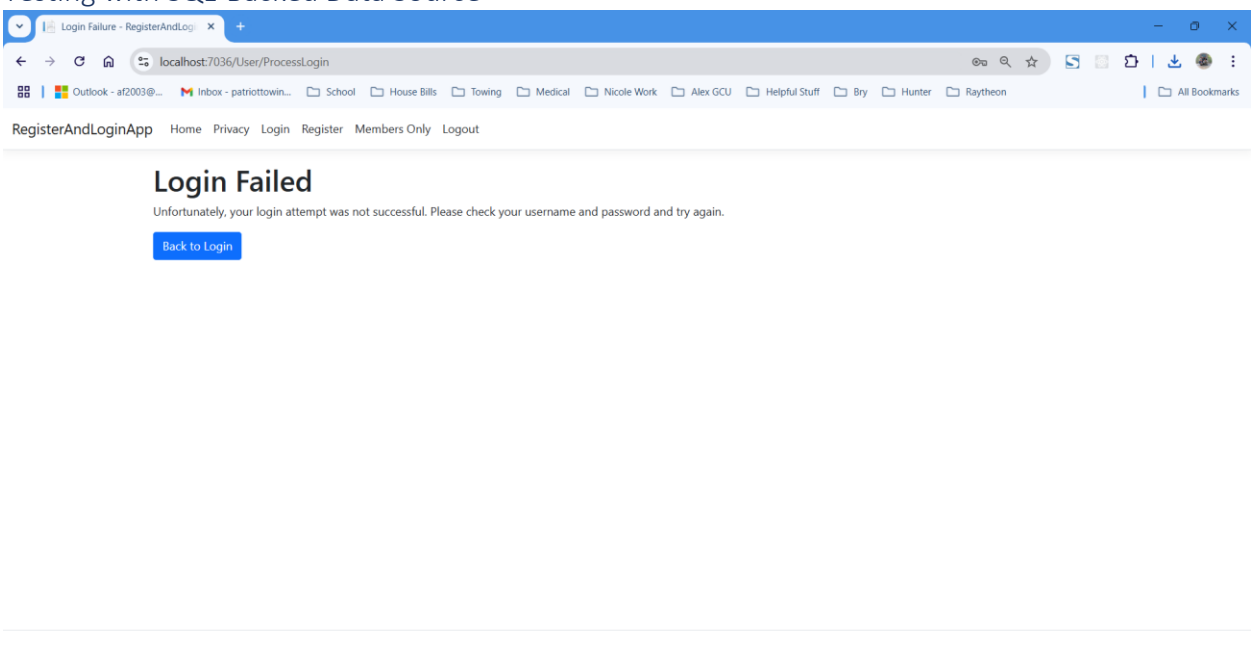
Figure 1 The UserController class is modified to accept an IUserManager instance through constructor injection, enabling dependency injection of different user manager classes.

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 The initial configuration in Program.cs injects UserDAO as the implementation of IUserManager, using AddSingleton to provide a consistent instance across the application.

Testing with SQL-Backed Data Source



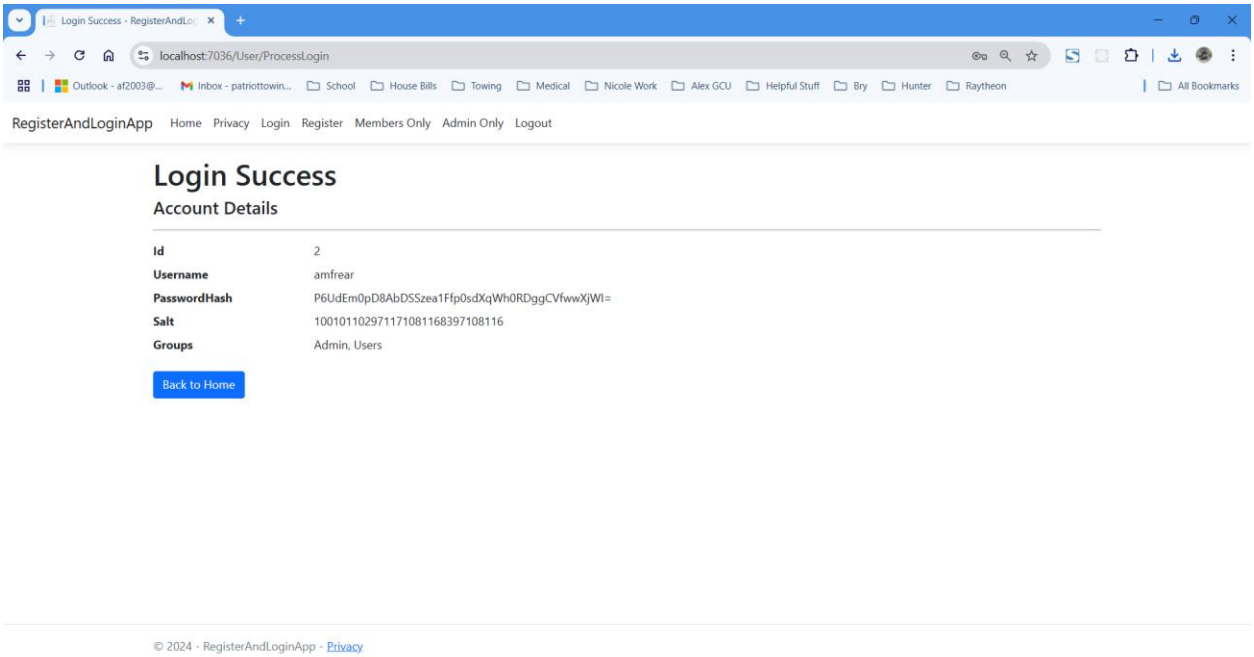


Figure 3 & 4 Initially, the application uses UserDAO, allowing login attempts against users stored in the SQL database. The application shows both successful and failed login attempts depending on user credentials.

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 The dependency injection configuration in Program.cs is updated to inject UserCollection as the IUserManager implementation, switching the data source to an in-memory collection of users.

Testing with In-Memory Data Source

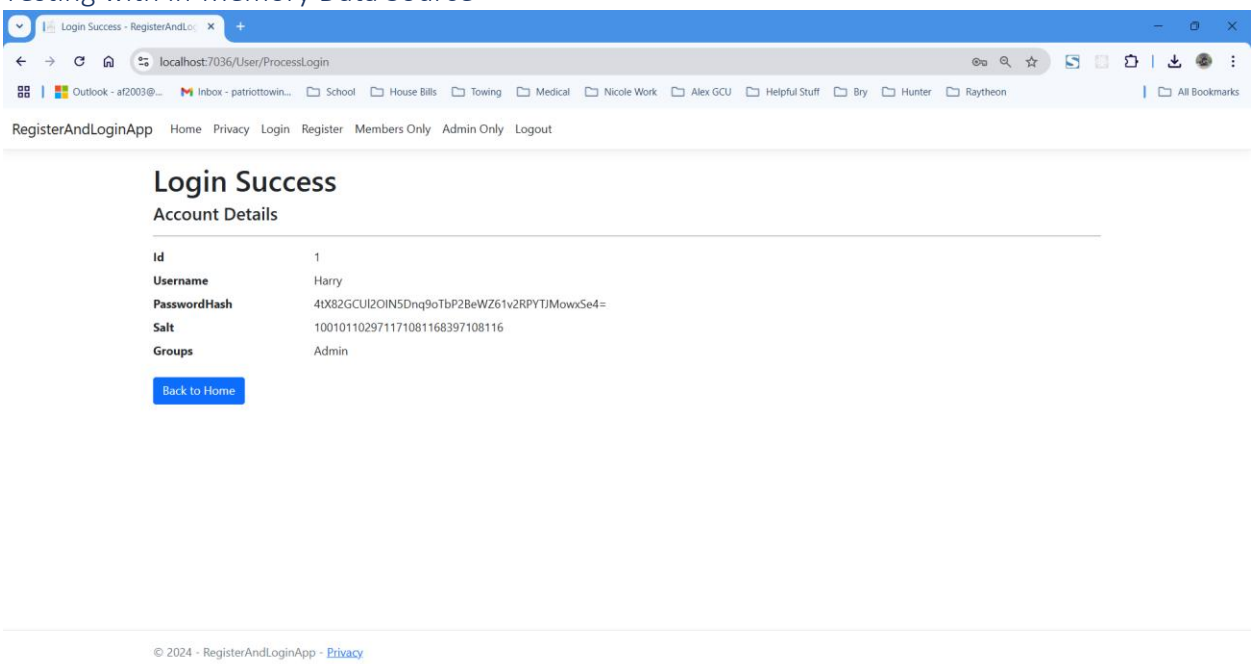


Figure 6 With UserCollection injected, the application allows login with in-memory data (e.g., the user "Harry"), showcasing dependency injection's flexibility by switching data sources without modifying the UserController.

Summary of Key Concepts (Part 1)

In Part 1 of Activity 4, I implemented dependency injection in the UserController class, allowing for flexible selection between data sources (UserDAO for SQL and UserCollection for in-memory data). By configuring Program.cs to inject IUserManager with either UserDAO or UserCollection, I demonstrated how dependency injection decouples the controller from a specific data implementation. This setup enabled easy data source switching and provided a practical understanding of dependency injection, enhancing the application's modularity and testability.

Part 2: Data Validation and Form Enhancements

Screenshots

Initial Validation Errors for Empty Fields

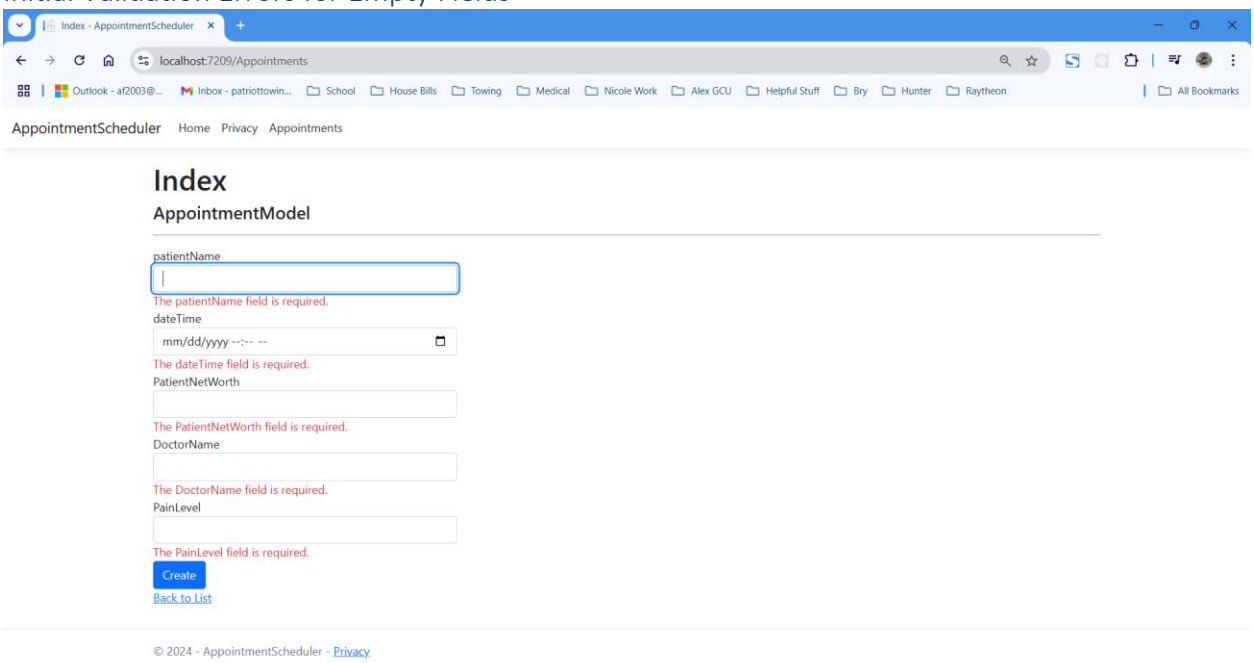


Figure 7 The form displays initial validation errors for required fields, ensuring that no field is left blank.

Custom Labels and Validation Errors

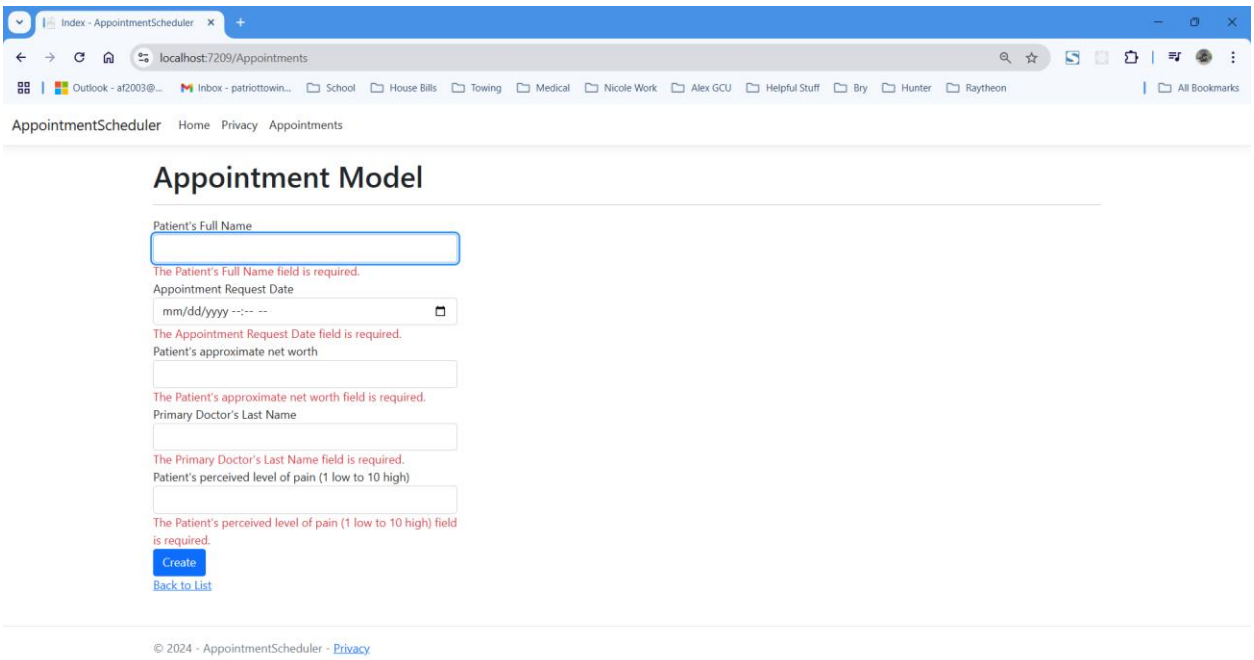


Figure 8 The appointment form shows customized labels for each field and displays validation error messages when fields are left empty.

Additional Validation Rules

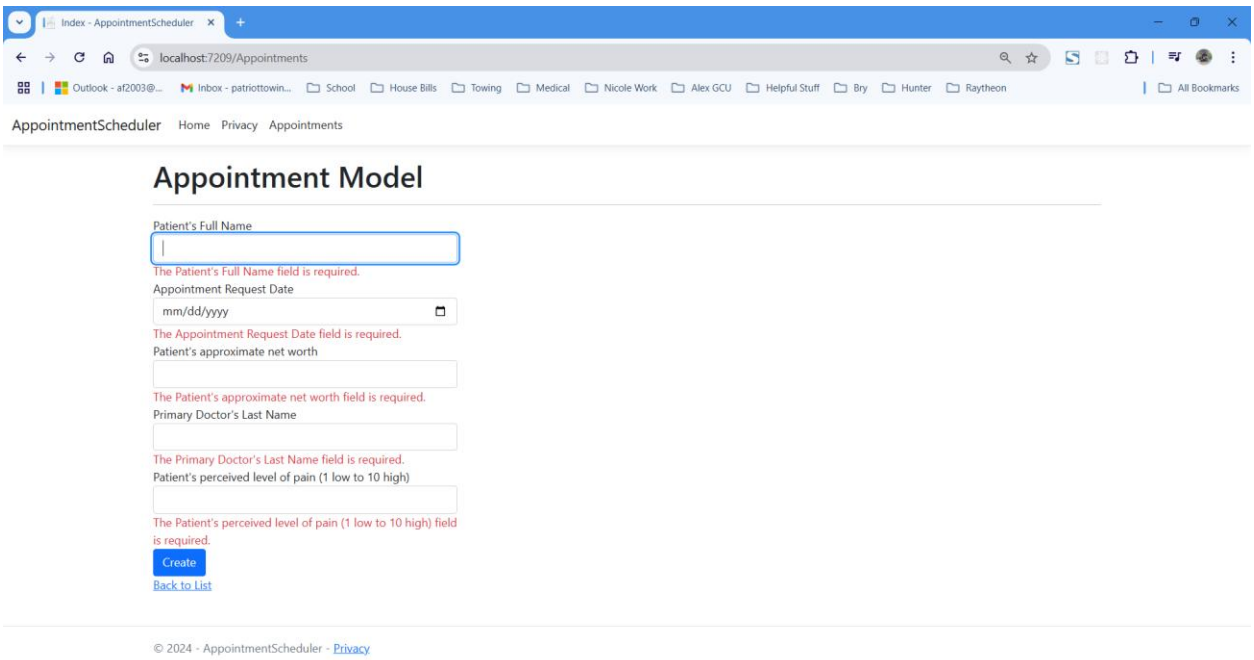


Figure 9 This screenshot shows the form with additional validation rules for net worth and pain level. Errors are shown if the net worth is below $90,000 or the pain level is 5 or below, in line with the business requirements.

Successful Submission Displaying Appointment Details

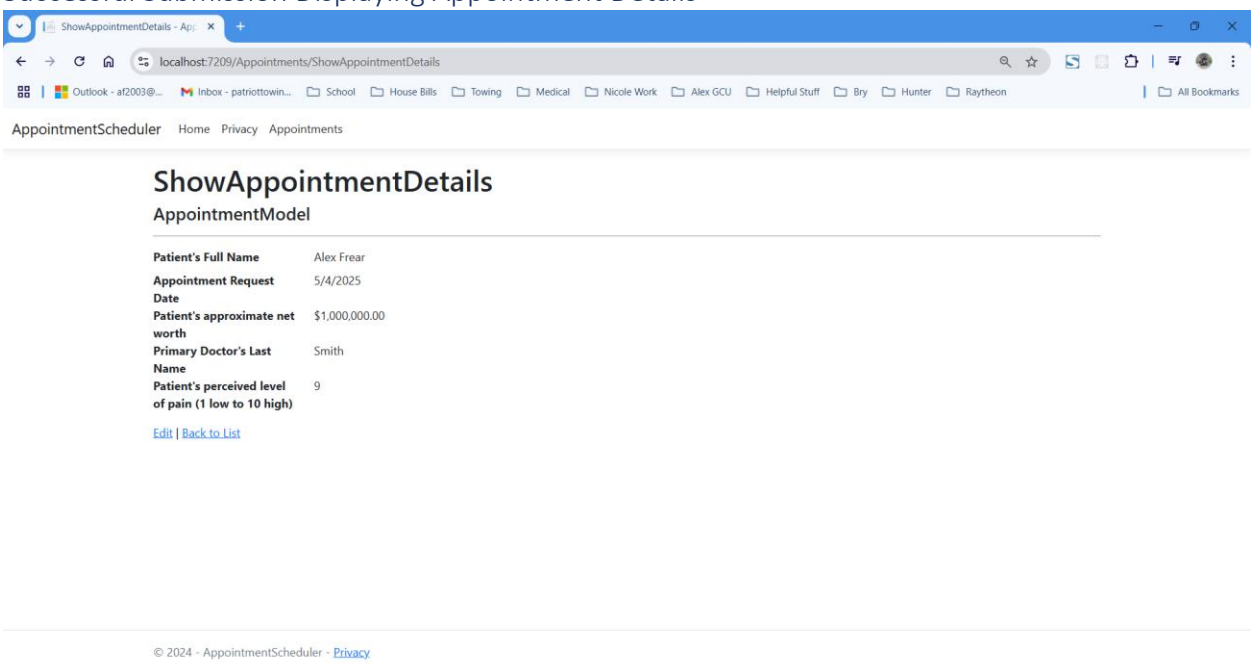


Figure 10 The final appointment details are displayed, showing a successful submission with all required information and restrictions met.

Validation Errors for Added Fields and Restrictions

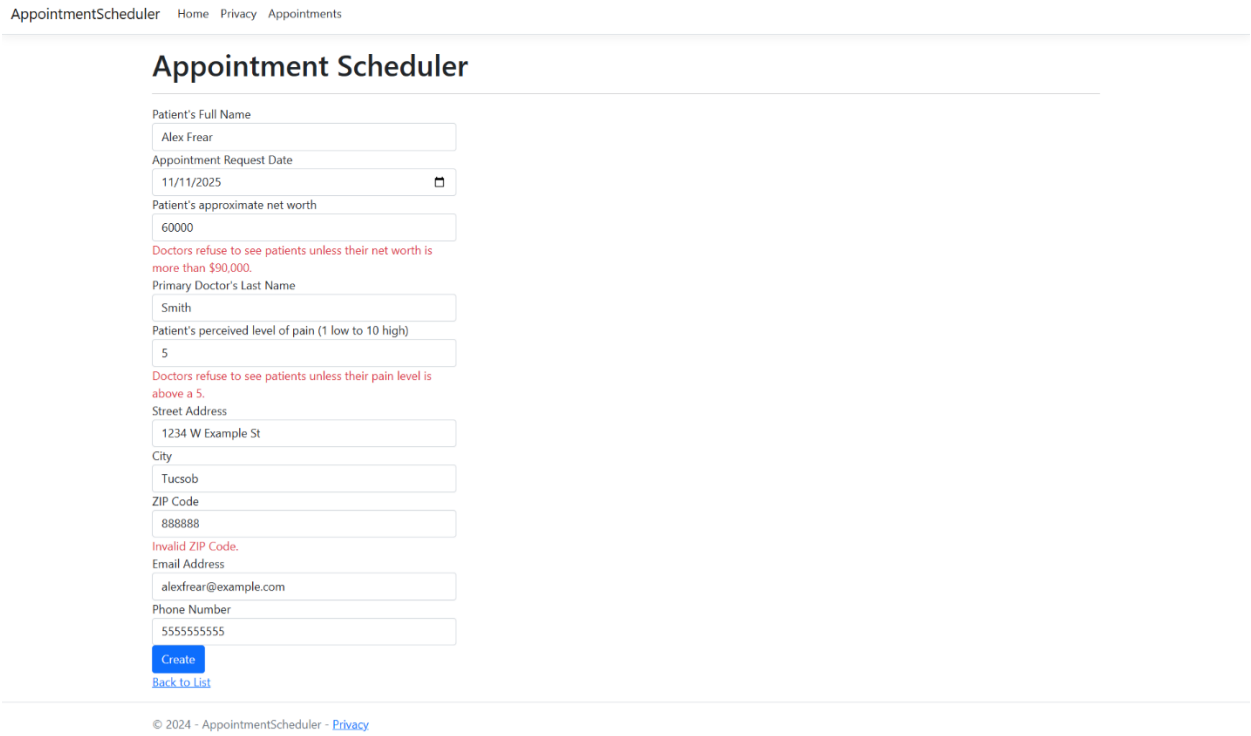


Figure 11 This screenshot demonstrates validation errors for new fields (such as ZIP code, email, and phone number) and specific restrictions on net worth and pain level. Errors are shown if the net worth is below $90,000 or the pain level is 5 or below, in line with the business requirements.

Successful Submission with New Fields and Restrictions

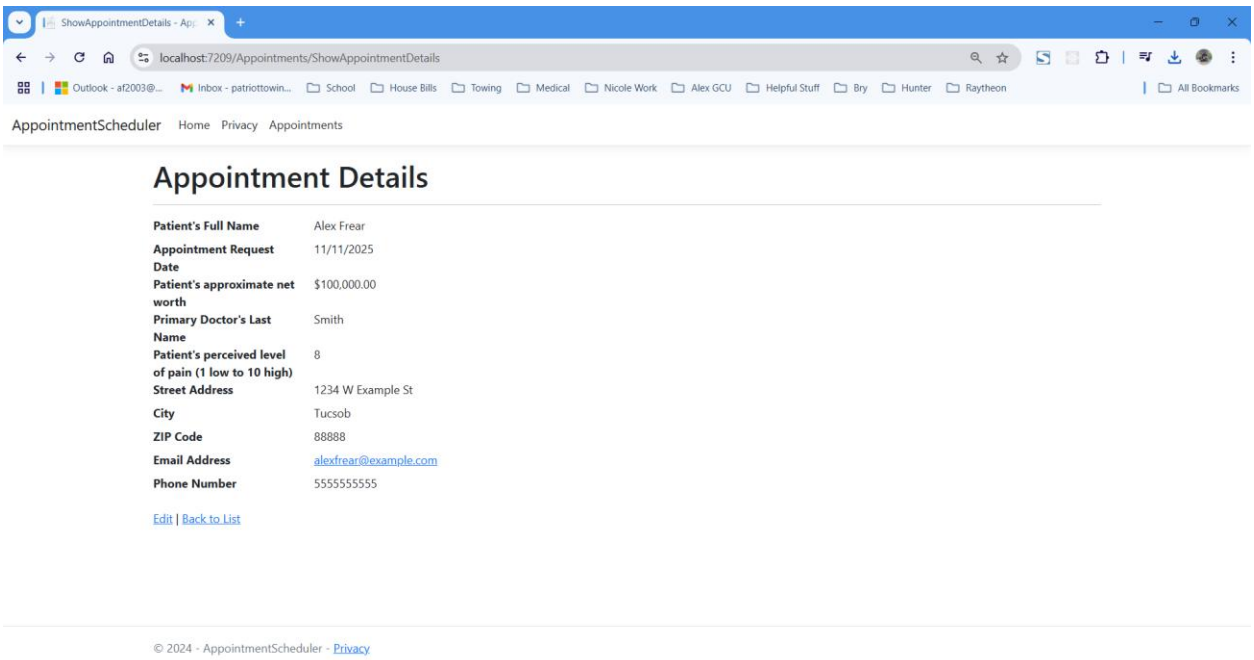


Figure 12 After filling in all required fields correctly, the form successfully submits, and the appointment details are displayed, including the new address fields and validation-compliant values.

Summary of Key Concepts (Part 2)

In Part 2 of Activity 4, I implemented additional validation rules in the AppointmentModel to ensure data integrity and enforce business logic. Custom display names were added to enhance form readability, and new fields for address, email, and phone were included. The model was further enhanced to reject appointments if the patient's net worth is below $90,000 or if the pain level is 5 or lower, as per the specified requirements. This part of the activity provided experience in creating custom validation messages, expanding form fields, and applying business rules to data entry.

Part 3: Button Grid Game Implementation with Success Message

Screenshots

Setting up the Button Grid with Initial State

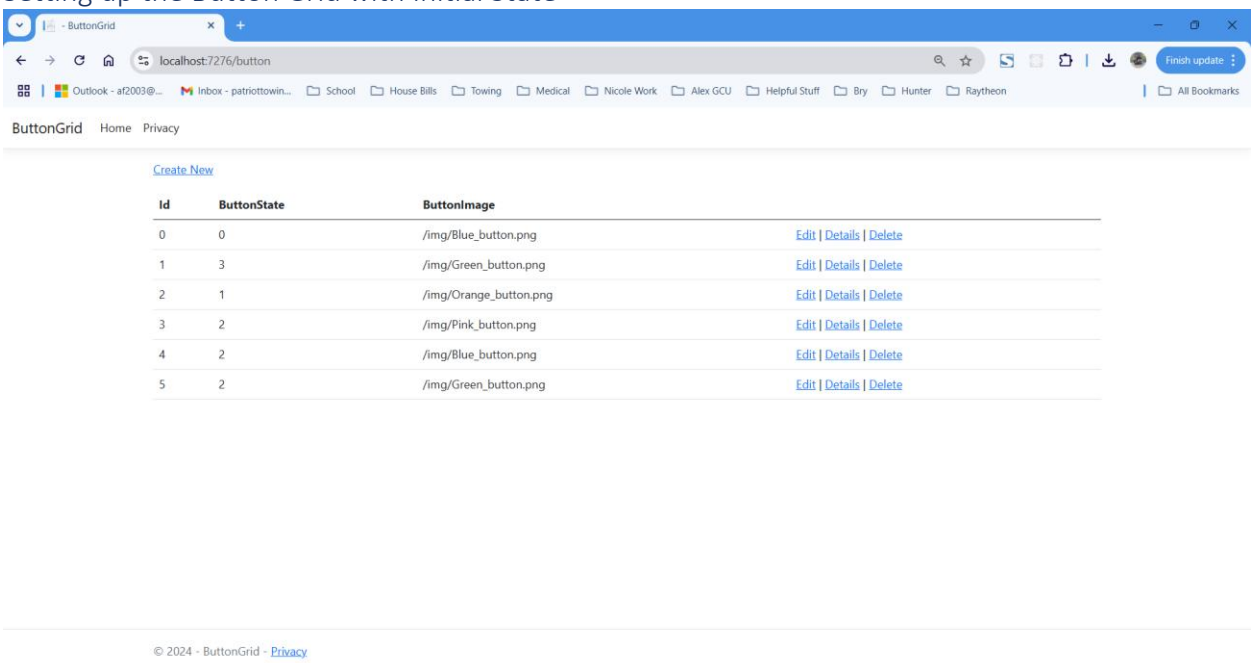


Figure 13 The initial setup of the button grid displays each button with its unique Id, ButtonState, and corresponding image based on its color.

Displaying the Button Grid with Timestamp

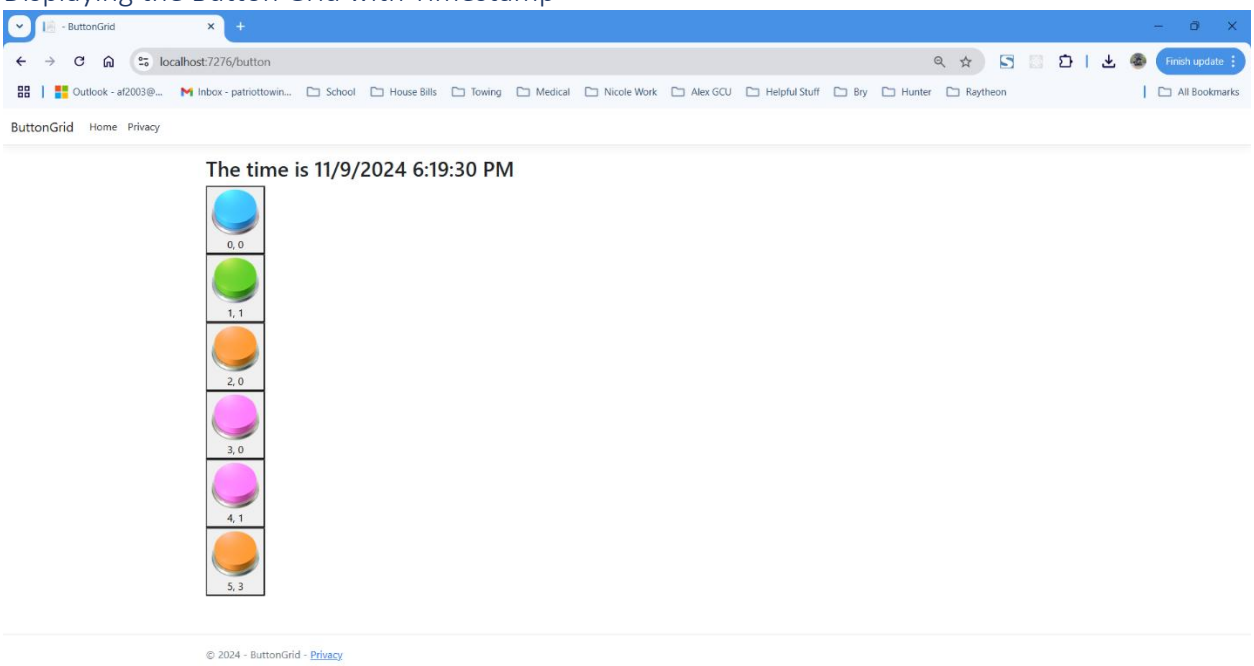


Figure 14 The grid view displays a live timestamp, providing a dynamic element to the page along with the clickable buttons.

Applying Flexbox for Grid Layout

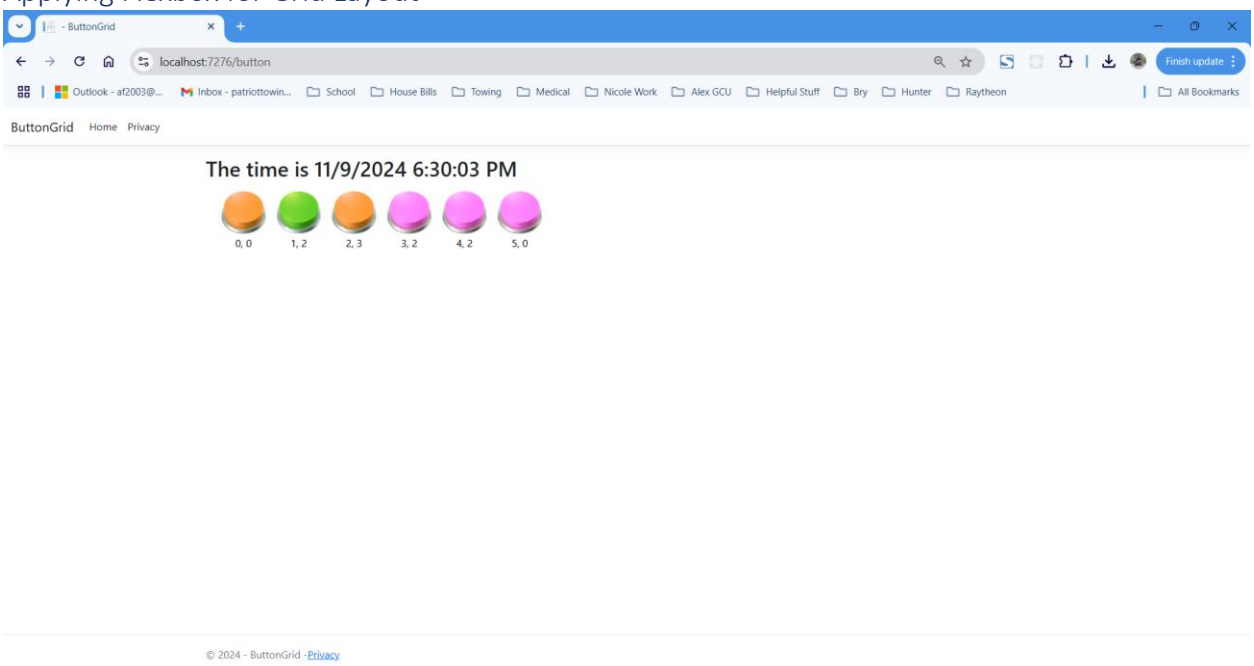


Figure 15 The button grid layout is enhanced using Flexbox, ensuring the buttons are aligned in rows. CSS styling is applied to improve visual consistency and spacing.

Inspecting HTML Elements

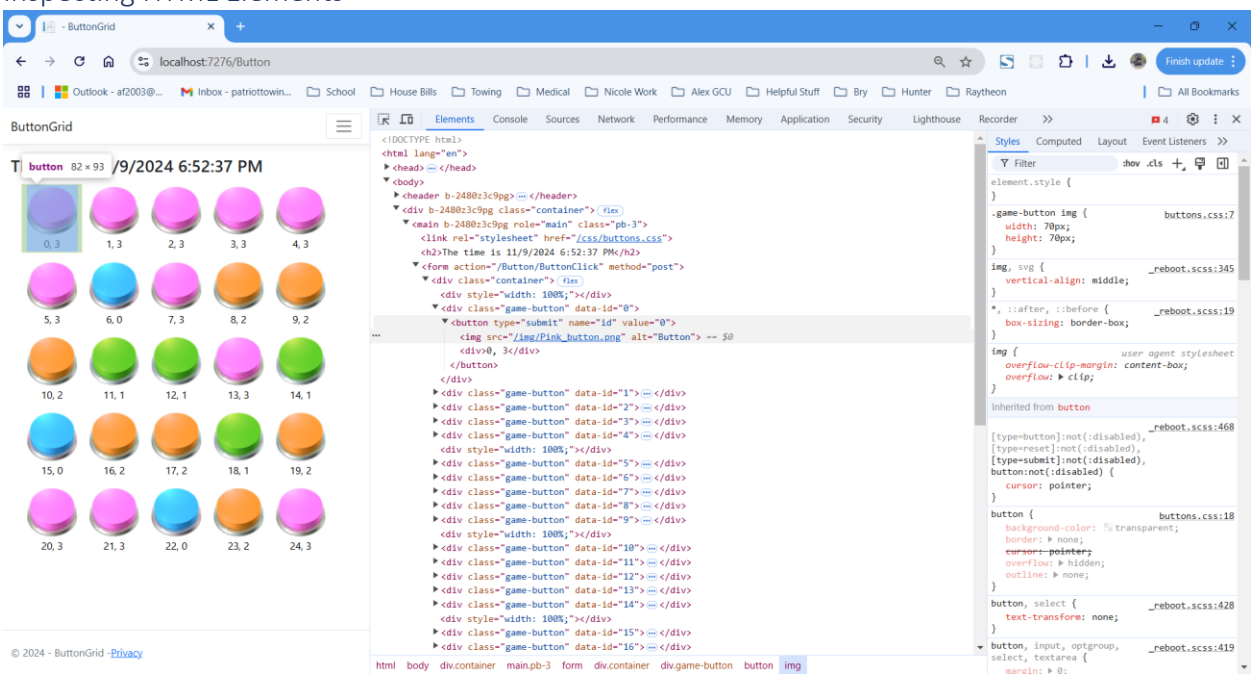


Figure 16 The HTML generated by the Razor view is inspected to confirm each button's structure, attributes, and functionality.

Game Success Message Display

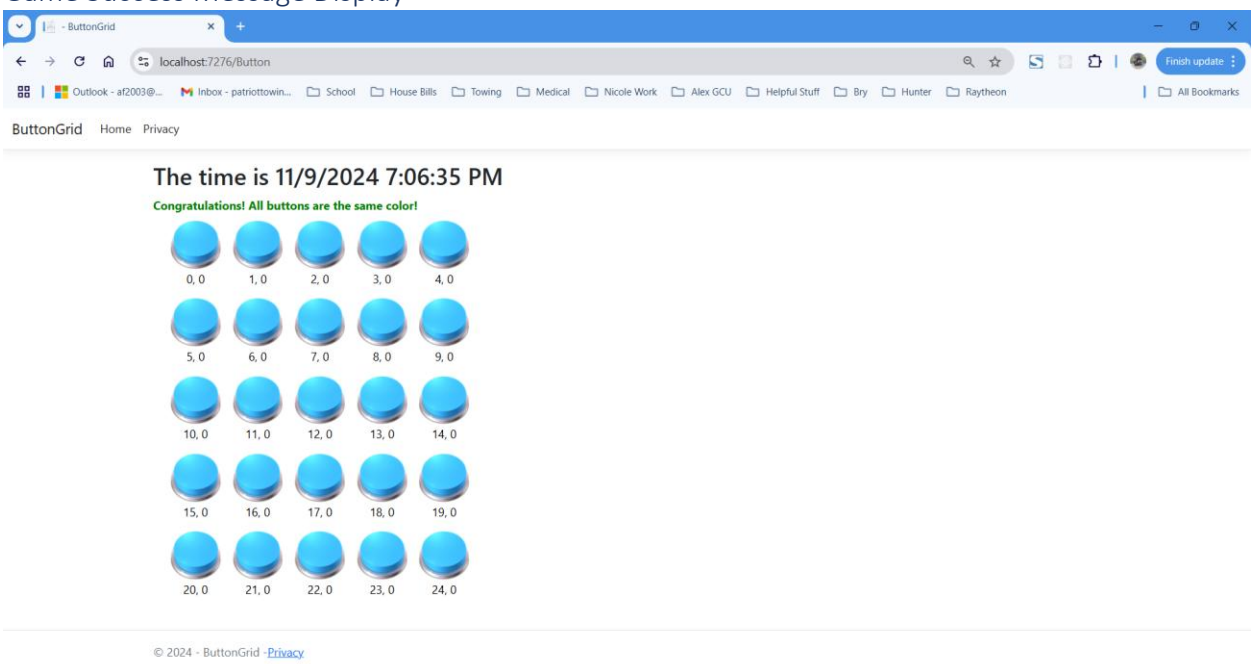


Figure 17 Once all buttons in the grid are set to the same color, a success message is displayed to the user, indicating that the game goal has been achieved.

Summary of Key Concepts (Part 3)

In Part 3, I created a button grid where each button changes state upon a click, cycling through a set of colors. Using Flexbox for layout ensured that the grid remained visually organized. Additionally, a success message is displayed when all buttons reach the same color, demonstrating simple game logic and enhancing user experience through visual feedback.