User.cs

```
public class User
    {
        public int Id { get; set; }
        public string Username { get; set; }
        public string Password { get; set; }
    }
```

UserRepository.cs

```
public interface IUserRepository
        {
            Task<User?> ValidateUser(string username, string password);
            Task<List<User>> GetAllUsers();
            Task<User?> GetUserById(int id);
            Task<User> AddUser(User user);
            Task<User> UpdateUser(User user);
            Task DeleteUser(int id);
        }

public class UserRepository : IUserRepository
{

    //private List<User> users = new List<User>
    //    {

    //        new User { Id = 1, Username = "admin", Password = "admin" },
    //        new User { Id = 2, Username = "user", Password = "user" },
    //        new User { Id = 3, Username = "Pranaya", Password = "Test@1234" },
    //        new User { Id = 4, Username = "Kumar", Password = "Admin@123" }
    //    };

    private readonly UserDBContext _context;

    public UserRepository(UserDBContext context)
    {
        _context = context;
    }

    public async Task<User?> ValidateUser(string username, string password)
    {
        await Task.Delay(100);
        //return users.FirstOrDefault(u => u.Username == username && u.Password ==
password);
        return _context.Users.FirstOrDefault(u => u.Username == username && u.Password ==
```

```
password);
    }

    public async Task<IEnumerable<User>> GetAllUsers()
    {
        await Task.Delay(100);
        //return users.ToList();
        return await _context.Users.ToListAsync();
    }

    public async Task<User?> GetUserById(int id)
    {
        await Task.Delay(100);
        //return users.FirstOrDefault(u => u.Id == id);
        return await _context.Users.FirstOrDefaultAsync(u => u.Id == id);
    }

    public async Task<User> AddUser(User user)
    {
        await Task.Delay(100);
        //if (users.Any(u => u.Id == user.Id))
        //{
        //    throw new Exception("User already exists with the given ID.");
        //}
        //users.Add(user);
        //return user;
        if (_context.Users.Any(u => u.Id == user.Id))
        {
            throw new Exception("User already exists with the given ID.");
        }
        _context.Users.Add(user);
        await _context.SaveChangesAsync();
        return user;
    }

    public async Task<User> UpdateUser(User user)
    {
        await Task.Delay(100);
        //var existingUser = await GetUserById(user.Id);
        //if (existingUser == null)
        //{
        //    throw new Exception("User not found.");
        //}
        //existingUser.Username = user.Username;
```

```
        //existingUser.Password = user.Password;
        //return existingUser;
        var existingUser = await _context.Users.FindAsync(user.Id);
        if(existingUser == null)
        {
            throw new Exception("User not found.");
        }
        existingUser.Username = user.Username;
        existingUser.Password = user.Password;
        _context.Entry(existingUser).CurrentValues.SetValues(user);
        await _context.SaveChangesAsync();
        return existingUser;
    }

    public async Task DeleteUser(int id)
    {
        //await Task.Delay(100);
        //var user = await GetUserById(id);
        //if (user == null)
        //{
        //    throw new Exception("User not found.");
        //}
        //users.Remove(user);
        var userr = await GetUserById(id);
        if(userr == null)
        {
            throw new Exception("User not found.");
        }
        _context.Users.Remove(userr);
        await _context.SaveChangesAsync();
    }
}
```

BasicAuthenticationHandler.cs

```
public class BasicAuthenticationHandler :
AuthenticationHandler<AuthenticationSchemeOptions>
    {
        private readonly IUserRepository _userRepository;

        public BasicAuthenticationHandler(
            IOptionsMonitor<AuthenticationSchemeOptions> options,
            ILoggerFactory logger,
```

```
        UrlEncoder encoder,
        IUserRepository userRepository)
        : base(options, logger, encoder)
    {
        _userRepository = userRepository;
    }

    protected override async Task<AuthenticateResult> HandleAuthenticateAsync()
    {

        if (!Request.Headers.ContainsKey("Authorization"))
        {
            return AuthenticateResult.Fail("Missing Authorization Header");
        }

        User? user;
        try
        {
            if (!AuthenticationHeaderValue.TryParse(Request.Headers["Authorization"], out var
authHeader))
            {
                return AuthenticateResult.Fail("Invalid Authorization Header Format");
            }
            var credentialBytes = Convert.FromBase64String(authHeader.Parameter ??
string.Empty);
            var credentials = Encoding.UTF8.GetString(credentialBytes).Split(':', 2);
            if (credentials.Length != 2)
            {
                return AuthenticateResult.Fail("Invalid Authorization Header Content");
            }

            var username = credentials[0];

            var password = credentials[1];

            user = await _userRepository.ValidateUser(username, password);
        }
        catch (FormatException)
        {
            return AuthenticateResult.Fail("Invalid Base64 Encoding in Authorization Header");
        }
        catch (Exception)
        {
            return AuthenticateResult.Fail("Error Processing Authorization Header");
```

```
        }
        if (user == null)
        {
            return AuthenticateResult.Fail("Invalid Username or Password");
        }

        var claims = new[] {
            new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
            new Claim(ClaimTypes.Name, user.Username),
        };

        var identity = new ClaimsIdentity(claims, Scheme.Name);

        var principal = new ClaimsPrincipal(identity);

        var ticket = new AuthenticationTicket(principal, Scheme.Name);

        return AuthenticateResult.Success(ticket);
    }

    protected override async Task HandleChallengeAsync(AuthenticationProperties properties)
    {
        Response.Headers["WWW-Authenticate"] = "Basic realm=\"BasicAuthenticationDemo\",
charset=\"UTF-8\"";

        Response.StatusCode = 401;

        await Response.WriteAsync("You need to authenticate to access this resource.");
    }
}
```

Program.cs

```
//builder.Services.AddSingleton<IUserRepository, UserRepository>();

builder.Services.AddScoped<IUserRepository, UserRepository>();

builder.Services.AddAuthentication("BasicAuthentication")
    .AddScheme<AuthenticationSchemeOptions,
BasicAuthenticationHandler>("BasicAuthentication", options => { });

builder.Services.AddDbContext<UserDBContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("ConStr")));

builder.Services.AddControllers()
```

```
.AddJsonOptions(options =>
{
    // This will use the property names as defined in the C# model
    options.JsonSerializerOptions.PropertyNamingPolicy = null;
});
```

UsersController.cs

```
[Authorize(AuthenticationSchemes = "BasicAuthentication")]
    [ApiController]
    [Route("api/[controller]")]
    public class UsersController : ControllerBase
    {

        private readonly IUserRepository _userRepository;


        public UsersController(IUserRepository userRepository)
        {

            _userRepository = userRepository;
        }

        [HttpGet]
        public async Task<ActionResult<IEnumerable<User>>> GetUsers()
        {

            var users = await _userRepository.GetAllUsers();

            return Ok(users);
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<User>> GetUser(int id)
        {
            var user = await _userRepository.GetUserById(id);
            if (user == null)
            {
                return NotFound("User not found.");
            }
            return Ok(user);
        }
```

```
[HttpPost]
public async Task<ActionResult<User>> CreateUser([FromBody] User user)
{
    try
    {
        var createdUser = await _userRepository.AddUser(user);

        return CreatedAtAction(nameof(GetUser), new { id = createdUser.Id }, createdUser);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}

[HttpPut("{id}")]
public async Task<ActionResult> UpdateUser(int id, [FromBody] User user)
{
    if (id != user.Id)
    {
        return BadRequest("ID mismatch in the URL and body.");
    }
    try
    {
        await _userRepository.UpdateUser(user);
        return NoContent();
    }
    catch (Exception ex)
    {
        if (ex.Message == "User not found.")
        {
            return NotFound(ex.Message);
        }
        return BadRequest(ex.Message);
    }
}

[HttpDelete("{id}")]
public async Task<ActionResult> DeleteUser(int id)
{
    try
    {
        await _userRepository.DeleteUser(id);
        return NoContent();
```

```
        }
        catch (Exception ex)
        {
            if (ex.Message == "User not found.")
            {
                return NotFound(ex.Message);
            }
            return BadRequest(ex.Message);
        }
    }
}



//Client App - .Net Core Console App
//Program

static HttpClient client = new HttpClient();
        static async Task Main(string[] args)
        {
            // Set the base address of the API.
            client.BaseAddress = new Uri("https://localhost:7215");

            // Clear any previously set request headers.
            client.DefaultRequestHeaders.Accept.Clear();

            // Add JSON to the Accept header to tell the server to send data in JSON format.
            client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
            // Set up basic authentication for all requests.
            var byteArray = Encoding.ASCII.GetBytes("admin:admin"); // Encoding the username
and password as ASCII.
            client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Basic",
Convert.ToBase64String(byteArray));
            try
            {
                // Perform a GET request to retrieve all users.
                Console.WriteLine("Getting all users...");
                var users = await GetAsync("api/users");
                Console.WriteLine(users);
                // Perform a POST request to create a new user.
                Console.WriteLine("Creating a new user...");
                var newUser = new { Id = 5, Username = "newuser", Password = "newpassword" };
                var postResult = await PostAsync("api/users", newUser);
```

```
            Console.WriteLine(postResult);
            // Perform a PUT request to update a user.
            Console.WriteLine("Updating a user...");
            var updatedUser = new { Id = 5, Username = "updateduser", Password =
"updatedpassword" };
            var putResult = await PutAsync("api/users/5", updatedUser);
            Console.WriteLine(putResult);
            // Perform a DELETE request to remove a user.
            Console.WriteLine("Deleting a user...");
            var deleteResult = await DeleteAsync("api/users/5");
            Console.WriteLine(deleteResult);
            // Wait for a key press before closing to see the results.
            Console.ReadKey();
        }
        catch (Exception e)
        {
            // Output any exceptions to the console.
            Console.WriteLine(e.Message);
        }
    }
    // Method for GET requests.
    static async Task<string> GetAsync(string path)
    {
        HttpResponseMessage response = await client.GetAsync(path);
        // Check if the request was successful and read the response content as a string.
        return response.IsSuccessStatusCode ? await response.Content.ReadAsStringAsync() :
$"Error: {response.StatusCode}";
    }
    // Method for POST requests.
    static async Task<string> PostAsync(string path, object value)
    {
        HttpResponseMessage response = await client.PostAsJsonAsync(path, value);
        // Check if the request was successful and read the response content as a string.
        return response.IsSuccessStatusCode ? await response.Content.ReadAsStringAsync() :
$"Error: {response.StatusCode}";
    }
    // Method for PUT requests.
    static async Task<string> PutAsync(string path, object value)
    {
        HttpResponseMessage response = await client.PutAsJsonAsync(path, value);
        // Return a success message or an error message depending on the request status.
        return response.IsSuccessStatusCode ? "Updated successfully." : $"Error:
{response.StatusCode}";
    }
```

```
        // Method for DELETE requests.
        static async Task<string> DeleteAsync(string path)
        {
            HttpResponseMessage response = await client.DeleteAsync(path);
            // Return a success message or an error message depending on the request status.
            return response.IsSuccessStatusCode ? "Deleted successfully." : $"Error:
{response.StatusCode}";
        }
```