Banking App Project

Completed by Celeste Ndlovu, 4 April 2025

**Project Summary: Celeste's Banking App**

**1. Project Overview & Goal:**

* Celeste's Banking App is a web application developed using ASP.NET Web Forms designed to simulate core banking functionalities for both regular customers and administrators.
* The primary goal was to create a functional, database-driven application demonstrating proficiency in backend C# development, database interaction with SQL Server, user authentication, CRUD operations (Create, Read, Update, Delete), transaction processing, and modern frontend design principles within the ASP.NET Web Forms framework.
* It serves as a practical example of building a multi-user system with distinct roles and permissions.

**2. Core Features:**

* **User Features:**
  + **Registration:** New users can create an account with personal details and credentials.
  + **Login:** Registered users can securely log in using their email and password.
  + **Profile Management:** Users can view their profile details and update editable information (like mobile number, email, password).
  + **Account Viewing:** Users can view a list of their associated accounts (e.g., Savings, Cheque) and their current balances.
  + **Transaction Viewing:** Users can view a history of transactions related to their accounts.
  + **Funds Transfer:**
    - **Internal:** Transfer funds between their *own* different accounts.
    - **External:** Transfer funds to another registered customer's specified account using the recipient's Customer ID and Account ID.
  + **Basic Transactions (Implied via Profile):** The user profile page includes functionality to simulate deposits/withdrawals directly via a transaction form linked to their accounts (though a dedicated deposit/withdrawal page might be a future improvement).
* **Admin Features:**
  + **Admin Login:** Secure login for administrators.
  + **Customer Management:** Admins can view, add, update, and delete customer records.
  + **Account Management:** Admins can view, add, update, and delete customer accounts.
  + **Transaction Viewing:** Admins can view a comprehensive list of *all* customer transactions within the system.

**3. Technology Stack:**

* **Backend:** C#
* **Framework:** ASP.NET Web Forms
* **Database:** Microsoft SQL Server
* **Frontend:** HTML, CSS, JavaScript (primarily via ASP.NET controls and Bootstrap components)
* **UI Framework:** Bootstrap 5 (for layout, components, and responsiveness)
* **UI Enhancements:**
  + Bootstrap Icons (for visual elements)
  + Custom CSS (for specific styling, branding, custom card designs, hero sections)
  + Google Fonts (e.g., Poppins, Raleway, Roboto for enhanced typography)
  + AOS (Animate On Scroll) library (for subtle page element entrance animations)

**4. Frontend Design & UI/UX (In Detail):**

* **Foundation:** Bootstrap 5 was heavily utilized as the core frontend framework.
* **Layout:** The Bootstrap grid system (container, row, col-\* classes) was used extensively to create responsive layouts that adapt to different screen sizes (desktops, tablets, mobiles).
* **Components:** Standard Bootstrap components provided base styling and functionality for:
  + **Navigation:** navbar (sticky, responsive toggler) with dropdowns.
  + **Forms:** form-control, form-select, form-label, form-check for styled and consistent input fields, dropdowns, labels, and checkboxes.
  + **Buttons:** btn, btn-primary, btn-success, btn-danger, etc., for clear calls to action. d-grid was used for full-width buttons in forms.
  + **Cards:** card, card-body, card-header, shadow-sm were used extensively to group related content (login forms, management options, profile sections, data displays).
  + **Tables:** table, table-striped, table-bordered, table-hover applied to asp:GridView for clean data presentation. table-responsive wrappers were used.
  + **Alerts:** alert, alert-danger, alert-success used for displaying user feedback messages (lblMessage).
  + **Tabs:** nav-tabs and tab-content used on the Transfers page for organizing different transfer types.
* **Consistency:** An ASP.NET Master Page (BankingApp.Master) ensured a consistent header (navbar) and footer across all pages.
* **Visual Appeal:**
  + **Bootstrap Icons** replaced basic images for cleaner, scalable iconography (e.g., user profile, transactions, management options).
  + **Custom CSS** (custom-banking.css, mystyles.css) was added to implement unique designs like the home page hero section, the custom styling for feature cards (service-card hover effects), and potentially override default Bootstrap styles for branding.
  + **Google Fonts** were integrated to provide a more modern typographic feel than default browser fonts.
  + **AOS library** added subtle fade/zoom animations (data-aos) as elements scrolled into view, enhancing the user experience.
* **Modernization:** The process involved refactoring older HTML structures (like <center>) to use modern Bootstrap utility classes (text-center, justify-content-center, spacing utilities like mb-3, py-4, g-3, gap-2) for better maintainability and alignment with Bootstrap best practices.

**5. Validation:**

* A two-pronged approach was used:
  + **Client-Side:** ASP.NET Validation Controls (RequiredFieldValidator, CompareValidator, RegularExpressionValidator) were implemented directly in the .aspx markup. These provide immediate feedback to the user in the browser (if configured correctly with Display="Dynamic" and proper CSS) and prevent unnecessary postbacks for simple input errors (e.g., empty fields, invalid formats, non-matching passwords). They were styled using Bootstrap classes (text-danger, small) for visual consistency.
  + **Server-Side:** Robust validation was performed in the C# code-behind:
    - Checking Page.IsValid before processing form data to ensure all ASP.NET validators passed.
    - Manually parsing and validating input (e.g., decimal.TryParse for amounts) as a secondary check.
    - Performing business logic checks (e.g., sufficient funds, account existence, ownership verification for external transfers) *before* executing database commands.

**6. Database Interaction:**

* **Database:** Microsoft SQL Server was used to store all application data.
* **Tables:** Key tables include Customer, Account, Transact, and Admin, linked via Foreign Keys (cusID in Account, accID in Transact). Computed columns (e.g., cusID as c0001) were used for user-friendly IDs.
* **Data Access:** Standard ADO.NET components were used in the C# code-behind:
  + SqlConnection: To establish a connection to the database (using a connection string stored in Web.config).
  + SqlCommand: To define and execute SQL queries (SELECT, INSERT, UPDATE, DELETE) and stored procedures (if any). Parameterized queries (cmd.Parameters.AddWithValue) were used to prevent SQL Injection vulnerabilities.
  + SqlDataReader: Used for reading results row-by-row from SELECT queries.
  + SqlDataAdapter & DataTable: Used to fill data into memory (DataTable) for binding to controls like asp:GridView.
  + ExecuteScalar: Used to retrieve single values (like checking existence or getting a balance).
  + ExecuteNonQuery: Used for executing commands that don't return data (INSERT, UPDATE, DELETE).
* **Transaction Management:** Critically, SqlTransaction was implemented in the funds transfer logic (btnInternalTransfer\_Click, btnExternalTransfer\_Click). This ensures **atomicity** – multiple database operations (debiting one account, crediting another, logging both transactions) either all succeed together or all fail and get rolled back, preventing data inconsistencies.

**7. Challenges Faced & Solutions:**

* **Frontend Modernization:** Making traditional ASP.NET Web Forms look modern and responsive was a challenge. **Solution:** Heavy adoption of Bootstrap 5, custom CSS, modern fonts, icons, and animations (AOS). Refactoring away from older HTML practices.
* **Validation Issues:**
  + Silent client-side validation failures (button seemingly unresponsive). **Solution:** Systematically checking the browser console for JavaScript errors, ensuring TextMode="SingleLine" on validated textboxes (especially for currency/numbers), verifying validator properties (ControlToValidate, Display="Dynamic"), temporarily disabling validation (CausesValidation="false"), and adding ValidationSummary controls for clearer error reporting.
  + Server-side validation logic errors (e.g., amount checks). **Solution:** Using breakpoints in C# to step through the code, verifying Page.IsValid is checked first, ensuring correct parsing (decimal.TryParse), and checking logical conditions carefully.
* **Event Handling:** Buttons not triggering server-side code. **Solution:** Verifying exact match (case-sensitive) between OnClick attribute in ASPX and method name in CS file, using breakpoints or temporary alerts to confirm handler execution.
* **Data Integrity During Transfers:** Ensuring that money wasn't lost or created if only part of a transfer completed. **Solution:** Implementing SqlTransaction in the C# code to wrap all related database updates and inserts, ensuring either full success (Commit) or full failure (Rollback).
* **State Management:** Ensuring data (like the logged-in user ID) persisted across postbacks. **Solution:** Using ASP.NET Session state (Session["cusID"]) and checking its validity on page loads.

**8. Potential Improvements:**

* **Enhanced Security:** Implement robust password hashing (e.g., using ASP.NET Identity libraries or BCrypt.Net), add Two-Factor Authentication (2FA), refine role-based authorization, implement measures against Cross-Site Scripting (XSS).
* **User Experience (UX):**
  + Use AJAX (via UpdatePanels or Page Methods/jQuery) for partial page updates (e.g., updating grids after transfer without a full page refresh).
  + Implement a password recovery/"Forgot Password" feature.
  + Provide more detailed and user-friendly error messages.
  + Add loading indicators for longer operations.
* **Functionality:** Add features like generating account statements, bill payments, loan applications, profile picture uploads, international currency handling, admin reporting dashboards.
* **Code Structure:** For larger applications, consider implementing architectural patterns like Repository or Service layers to separate data access and business logic from the UI code-behind. Explore Dependency Injection.
* **Testing:** Implement unit tests for business logic and integration tests for database interactions.
* **Deployment:** Consider proper deployment configurations, error logging (e.g., ELMAH), and performance optimization.

This summary provides a solid overview of your project's capabilities, technical implementation, and potential future directions.