AwesomeGIC Bank – Console Banking Application

A simple console-based banking system demonstrating basic account operations with **persistent storage** using SQLite.

Built with **.NET** (C#), clean architecture, dependency injection, and **Entity Framework Core + SQLite**.

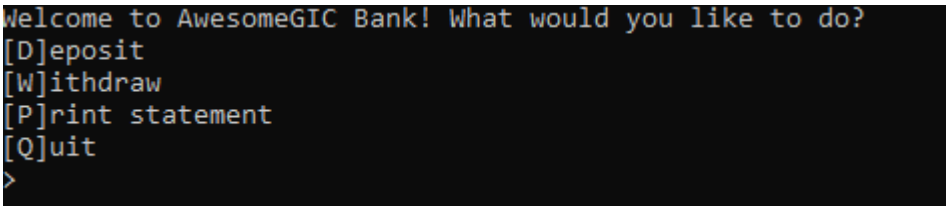
Table of Contents

Features

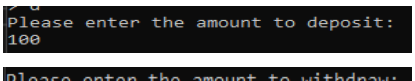
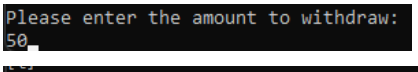
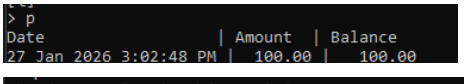
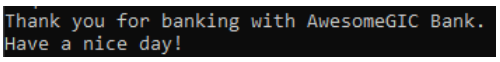
- Deposit and withdraw money
- Print formatted account statement (date | amount | balance)
- **All transactions are automatically saved to a SQLite database**
- Data persists between application restarts
- Case-insensitive input (D/d, W/w, P/p, Q/q)
- Basic validation and error messages

User Manual – How to Use the Application

Run the app → see the welcome screen:



Commands at a glance

Key	Action	Result / Visual
`D`/`d`	Deposit	
`W`/`w`	Withdraw	
`P`/`p`	Print statement	
`Q`/`q`	Quit	

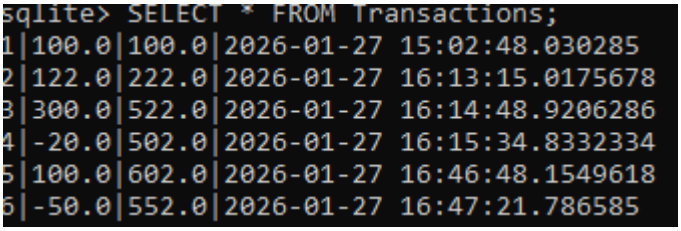
Key feature – Transaction Persistence in SQLite

Every successful deposit or withdrawal is **immediately saved** to the SQLite database file `awesome_gic_bank.db` (created automatically in the application directory).

Behavior:
- Close and restart the application → your transaction history is still there
- Print statement (`P`) shows all past transactions loaded from the database
- No need to manually save or export – persistence is automatic

Visual confirmation (for developers / curious users):

After performing several operations:



Technical Overview

- Language / Runtime: C# / .NET (8 / 9 / 10 compatible)
- UI: Console
- Architecture: Clean architecture + Dependency Inversion
- Persistence: **Entity Framework Core** + **SQLite**
- Dependency Injection: Microsoft.Extensions.DependencyInjection
- Testing: xUnit + FluentAssertions

Architecture & Design

Presentation (Console UI)
↓
Application Services
↓
Domain (Business Logic)
↕
Infrastructure (EF Core + SQLite)

Main Classes & Responsibilities

Class/Interface	Layer	Purpose
`Program`	Presentation	Entry point, composition root, main command loop
`ConsoleUserInterface`	Presentation	Console I/O implementation
`IBankAccountService`	Application	Account operations contract

| `BankAccountService` | Domain | Balance logic, transaction creation & validation
| `Transaction` | Domain | Value object (Date, Amount, BalanceAfter)
| `IStatementPrinter` | Application | Statement formatting contract
| `ConsoleStatementPrinter` | Presentation | Console table output
| `IPersistenceService` | Application | Save/load account state contract
| `EfPersistenceService` | Infrastructure | SQLite persistence via EF Core
| `BankDbContext` | Infrastructure | EF Core context with `DbSet<Transaction>`

Persistence & Database

Core feature: SQLite persistence

- Database file: `awesome_gic_bank.db` (created automatically)
- Table: `Transactions` (keyless / append-only design)
- Columns: `Date` (TEXT), `Amount` (NUMERIC), `BalanceAfter` (NUMERIC)
- Every deposit/withdraw calls `SaveAsync` → changes are persisted instantly
- On startup: transactions are loaded from DB into memory
- Uses **Entity Framework Core** with SQLite provider
- Migrations managed via `dotnet ef` CLI

Note: Currently uses a keyless entity (no primary key).
For production use, consider adding a surrogate `Id` column to prevent potential duplicate inserts on reload.

How to Run the Application

Bash

```
dotnet restore
dotnet build
dotnet run --project AwesomeGICBank1
```

Reset database (if needed):

```
dotnet ef database drop --project AwesomeGICBank1 --force
dotnet ef database update --project AwesomeGICBank1
```

How to Build & Test

```
dotnet build
dotnet test
```

Migrations

```
dotnet ef migrations add SomeChange --project AwesomeGICBank1
dotnet ef database update --project AwesomeGICBank1
```