Dr. Rajkumar, 2026, 14:3
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology

An Open Access Journal

Expense Tracker Web Application: Design and Development Using Angular, ASP.NET Core, and MySQL

Dr. Rajkumar[1], Vipin Kushwaha[2], Vasu Aggarwal[3], Vishal Singh[4], Ujjawal Charnotia[5]

[1]Assistant Professor Quantum University Roorkee

[2,3,4,5]BCA Scholar of Department of Computer Applications

Abstract- Effective personal financial management is a critical life skill that many individuals struggle to practice consistently. Conventional approaches such as paper ledgers and basic spreadsheet files are error-prone, difficult to analyze, and inadequate for modern financial needs. This paper presents the design, implementation, and evaluation of a web-based Expense Tracker Application intended to help users record, categories, and visualize their daily financial transactions in a structured and user-friendly manner. The proposed system is built on a three-tier architecture comprising Angular 14 on the client side, ASP.NET Core on the server side, and a MySQL relational database for persistent data storage. Communication between the front-end and back-end layers is facilitated through RESTful Application Programming Interfaces (APIs) transmitted over the Hypertext Transfer Protocol (HTTP). The application supports secure user authentication using JSON Web Tokens, category-based expense classification covering food, transportation, entertainment, utilities, and shopping, and dynamic report generation through interactive charts and tabular summaries. The system was designed with responsiveness in mind to perform consistently across desktop computers, laptops, tablets, and mobile devices. The study adopts the Software Development Life Cycle (SDLC) model encompassing requirements analysis, system design, implementation, and systematic testing. Evaluation results demonstrate that the proposed application significantly simplifies personal budgeting, enhances financial awareness, and addresses key limitations of commercially available tools. The paper further discusses the system architecture, database design, API structure, security mechanisms, and future enhancement directions.

Keywords: Expense Tracker, Web Application, Angular 14, ASP.NET Core, MySQL, Personal Finance Management, RESTful API, JWT Authentication, Software Development Life Cycle, Responsive Design.

I. INTRODUCTION

Financial transactions are an inseparable part of everyday life. Individuals routinely spend money on groceries, commuting, utility bills, education, and leisure activities. Despite the frequency of these transactions, a substantial number of people do not maintain systematic records of their expenditures. The absence of such records contributes to poor

budgeting, uninformed financial decisions, and unexpected monetary shortfalls at the end of monthly cycles [1]. Studies have shown that individuals who track their expenses are statistically more likely to meet savings targets and avoid high-interest consumer debt than those who do not.

Traditional expense management methods, including handwritten notebooks and basic spreadsheet applications, are widely used but inherently limited. They are time-consuming to maintain, prone to transcription errors, and incapable of generating insightful analytical summaries without significant manual effort. Furthermore, they lack the interactivity and visual reporting features that modern users have come to expect from digital tools [2]. The process of manually totalling columns and drawing inferences about spending patterns is both tedious and cognitively demanding, reducing the likelihood that users will engage with their financial data on a regular basis.

Digital financial management platforms have emerged to address these shortcomings. However, many commercially available tools require paid subscriptions to unlock essential features, or they are laden with functionalities that ordinary users do not require, making the interfaces complex and the learning curve unnecessarily steep [3]. Popular services such as Mint, YNAB (You Need A Budget), and Walnut offer comprehensive feature sets but restrict export capabilities, advanced analytics, or multi-device synchronisation behind paywalls. Moreover, users who are conscious of data privacy are often reluctant to entrust sensitive financial records to third-party cloud platforms whose data-handling policies may be opaque.

The present work proposes and implements a lightweight, open-source Expense Tracker Web Application that allows users to register, authenticate, record transactions, classify them by category, and review summarized visual reports. The system is architected as a three-tier application: Angular 14 handles the presentation layer, ASP.NET Core manages the business logic and API layer, and MySQL provides persistent relational storage. This technology stack was deliberately chosen because all constituent components are freely available, well supported by large developer communities, and widely taught in academic computing programmes.
The primary contributions of this paper are: (i) a detailed description of the system architecture and design rationale for a practical personal finance application; (ii) an account of the security mechanisms employed to protect user credentials and financial data; (iii) a feature comparison between the proposed system and commercial alternatives; and (iv) a discussion of testing outcomes and identified areas for future enhancement. The remainder of the paper is structured as follows. Section II reviews existing literature and related systems. Section III describes the methodology, architecture, and system design. Section IV presents results and discussion. Section V concludes the paper and outlines future directions.

II. LITERATURE REVIEW

Research into digital personal finance management has grown considerably over the past decade. Scholars and practitioners have highlighted both the behavioral and technological dimensions of the problem, arriving at the consensus that automated, category-aware tools are more effective at improving financial literacy than manual record-keeping methods [4]. Behavioral economics research demonstrates that visual feedback on spending—particularly when presented as proportional charts—activates loss-aversion mechanisms that motivate users to moderate discretionary expenditures more effectively than text-based summaries.

Studies examining mobile-based budgeting applications have reported that users who engage with digital expense trackers demonstrate greater awareness of their spending patterns and are more likely to adjust their behavior in response to analytical feedback [2]. A longitudinal study conducted over six months found that participants using a category-based mobile tracker reduced their discretionary spending by an average of 14 percent compared to a control group maintaining paper records. Visual representations such as pie

charts and bar graphs were identified as particularly effective in communicating complex financial data in an accessible format that required minimal financial literacy to interpret correctly.

From an architectural perspective, existing web-based financial systems commonly adopt the Model-View-Controller (MVC) design pattern and RESTful API architectures to decouple front-end presentation from back-end business logic. This separation of concerns enhances maintainability, supports independent scaling of application tiers, and facilitates future feature additions without requiring wholesale redesign [5]. Micro services architectures have also been explored for larger-scale financial platforms, though the associated operational complexity is disproportionate for personal-scale applications of the kind described in this paper.

Angular, as a component-based Typescript framework developed and maintained by Google, has been adopted in several enterprise-grade financial dashboards owing to its robust dependency injection system, reactive forms module, built-in HTTP client, and comprehensive testing utilities [3].

Its use of TypeScript adds static type safety that reduces runtime errors in data-handling logic—a particularly valuable characteristic in financial applications where data integrity is paramount. Competing frameworks such as React and Vue.js offer similar capabilities but lack Angular's opinionated project structure, which accelerates onboarding for teams familiar with the framework's conventions.

ASP.NET Core has similarly been adopted in server-side financial applications because it offers a middleware-based request pipeline, built-in support for JWT bearer authentication, seamless integration with Entity Framework Core for object-relational mapping, and cross-platform deployment on Linux, macOS, and Windows servers [5]. Its performance benchmarks consistently rank among the highest for managed-runtime web frameworks, making it suitable for API workloads that require low-latency responses to concurrent user requests.

Relational databases, particularly MySQL and PostgreSQL, remain the dominant choice for financial data storage owing to their ACID (Atomicity, Consistency, Isolation, Durability) compliance, mature indexing mechanisms, and robust support for complex join queries required in financial aggregation operations [6]. NoSQL alternatives such as MongoDB are occasionally employed for high-volume transaction logging but are less suitable for the structured querying patterns associated with budget reporting.

Despite these advances, a survey of available tools reveals persistent gaps. Many existing applications require cloud subscriptions, expose overly complex interfaces to casual users, or lack transparency in data handling. Several academic prototypes have been reported in the literature but are not made publicly available, limiting their practical utility. The proposed system addresses each of these shortcomings by delivering a self-hostable, open-source solution with a minimalist interface centred on core expense-tracking functionality, while remaining extensible enough to accommodate future enhancements.

III. METHODOLOGY

The development of the Expense Tracker Web Application followed the classical Software Development Life Cycle (SDLC), which provides a structured framework for guiding a system from initial conception through deployment and maintenance. The SDLC was chosen over agile sprint-based methodologies because the project scope and requirements were sufficiently well defined at the outset to permit upfront design, and the team size was small enough that iterative ceremony overhead was not justified. The five principal phases executed were: requirement analysis, system design, implementation, testing, and deployment.

A. Requirement Analysis

During the requirement elicitation phase, the core functional and non-functional requirements were identified through examination of common personal finance management workflows. Functional requirements established that the system must: (i) allow users to create accounts with validated credentials and authenticate securely; (ii) support addition, editing, deletion, and retrieval of financial transaction records; (iii) permit assignment of each transaction to a predefined or user-defined category; (iv) enable filtering of historical records by date range, category, or amount; and (v) generate graphical summaries of spending behaviour on demand.

Non-functional requirements specified that API response times must remain below 500 ms for typical queries on datasets of up to 10,000 transactions per user; the user interface must be fully functional on viewport widths between 360 px and 1920 px; user passwords must never be stored or transmitted in plaintext; and the entire system must be deployable on commodity hosting infrastructure without proprietary dependencies.

B. System Architecture and Design

The system was designed following three-tier client-server architecture, as illustrated conceptually in Fig. 1. The presentation tier comprises an Angular 14 single-page application (SPA) that is served as static assets and executes entirely within the user's browser. The logic tier is an ASP.NET Core 6.0 Web API that exposes RESTful endpoints and enforces business rules. The data tier is a MySQL 8.0 relational database accessed through Entity Framework Core 6.0, which provides an object-relational mapping layer that abstracts raw SQL from application code.

The database schema is normalised to the third normal form and centres on three primary entities: Users, Categories, and Transactions. The Users table stores hashed credentials, display names, and registration timestamps. The Categories table holds both system-defined defaults and user-created categories, distinguished by an is_system boolean flag. The Transactions table captures individual records including amount (stored as DECIMAL (10, 2) to avoid floating-point rounding errors), description, transaction date, and foreign keys referencing both Users and Categories. A summary of the schema is presented in Table I.

Table I: Database Schema Summary

Table	Key Columns	Relationships
Users	user_id (PK), email, password_hash, created_at	—
Categories	category_id (PK), name, is_system, user_id (FK)	Belongs to Users
Transactions	txn_id (PK), user_id (FK), category_id (FK), amount, description, txn_date	Belongs to Users, Categories

API endpoints are organised into three controllers. The AuthController exposes routes for user registration, login, and token refresh. The CategoryController provides full CRUD operations on expense categories. The TransactionController handles CRUD on individual transactions and additionally exposes aggregation endpoints used by the reporting module, which perform GROUP BY queries in MySQL to calculate per-category and per-month totals. All protected routes are secured by ASP.NET Core's JWT bearer middleware, which validates token signatures, expiry claims, and issuer metadata on every incoming request.

C. Front-End Implementation

The Angular front-end is structured into four lazy-loaded feature modules to minimise initial bundle size: AuthModule (login and registration forms), DashboardModule (overview widgets and summary cards), TransactionModule (transaction list, create, and edit views), and ReportModule (chart and table-based spending reports). Lazy loading ensures that users who navigate only to the login page do not download the JavaScript bundles for reporting or transaction management until those routes are accessed.

Reactive Forms are used throughout the application for data entry because they provide a model-driven approach to form validation that is more testable

and composable than Angular's alternative template-driven forms. Custom validators enforce business rules such as non-negative amounts and date fields that do not exceed the current system date. Route guards (CanActivate) intercept navigation to authenticated areas and redirect unauthenticated users to the login screen.

The HttpClient service communicates with the ASP.NET Core API using standard HTTP verbs: GET for retrieval, POST for creation, PUT for full updates, PATCH for partial updates, and DELETE for removal. An HTTP interceptor automatically attaches the JWT access token as an Authorization header to every outgoing request, relieving individual service classes from the responsibility of token management. A separate error-interceptor catches 401 unauthorized responses and triggers a token-refresh flow before retrying the failed request.

Chart rendering is handled by Chart.js, integrated via the ng2-charts Angular wrapper. Pie charts display the proportion of total expenditure attributable to each category for a selected period. Bar charts present monthly expenditure totals across a configurable date range, enabling users to identify seasonal or irregular spending patterns. All charts are dynamically updated when the user modifies filter parameters without requiring page navigation.

D. Back-End Implementation

The ASP.NET Core back-end follows the repository pattern to abstract data-access logic from controller logic. Each entity has a corresponding interface (e.g., ITransactionRepository) and a concrete Entity Framework Core implementation. The dependency injection container wires interfaces to implementations at application startup, enabling controllers to be tested in isolation using mock repositories.

Password security is implemented using BCrypt with a work factor of 12, which balances security against the computational cost of authentication. BCrypt is a deliberately slow hashing algorithm designed to resist brute-force and rainbow-table attacks; the work factor can be increased as hardware improves without requiring changes to the stored hash format. Plaintext passwords are never persisted to the database or written to log files.

JWT tokens are issued with a 60-minute expiry and signed using HMAC-SHA256 with a 256-bit secret key stored as an environment variable outside the application codebase. Refresh tokens with a 7-day validity are stored in the database and rotated on each use to mitigate the risk of token theft. The combination of short-lived access tokens and rotating refresh tokens balances security with user convenience.

CORS (Cross-Origin Resource Sharing) policy is configured to allow requests only from the known Angular application origin, preventing unauthorised third-party websites from making authenticated API calls on behalf of logged-in users. Input validation is enforced using ASP.NET Core's built-in data annotations and FluentValidation, which reject malformed requests before they reach repository or database layers.

E. Testing

A multi-layer testing strategy was adopted to validate correctness at each tier of the application. Unit tests for Angular components and services were written using the Jasmine testing framework and executed by the Karma test runner. These tests verified component rendering logic, form validation behaviour, and service method outputs using mock HTTP responses rather than live API calls.

Back-end unit tests were authored using the xUnit framework with Moq as the mocking library. Tests were written for each controller action and repository method, verifying that correct HTTP status codes were returned for valid and invalid inputs, and that repository methods were invoked with the expected parameters. An in-memory SQLite database, configured through Entity Framework Core's UseInMemoryDatabase provider, was used for integration tests that exercised the full data-access stack without requiring a running MySQL instance.

System-level acceptance tests were executed manually against a locally deployed instance of the complete application. Test cases covered user registration with duplicate email detection, login with correct and incorrect credentials, transaction creation with boundary values, category filter accuracy, and report data consistency between API aggregation results and chart visual output. Cross-browser compatibility was verified on Google Chrome 112, Mozilla Firefox 113, and Microsoft Edge 113.

IV. RESULTS AND DISCUSSION

The completed application was evaluated against the functional and non-functional requirements established during the analysis phase. All core features were successfully implemented and verified through the testing procedures described in Section III. Table II presents a summary of feature implementation status, and Table III provides a comparative analysis against two widely used commercial alternatives.

Table II: Feature Implementation and Verification Summary

Feature	Requirement	Status
User Registration & Login	Functional	Implemented
JWT Authentication	Security	Implemented
Transaction CRUD	Functional	Implemented
Category Management	Functional	Implemented
Date & Category Filtering	Functional	Implemented
Pie & Bar Chart Reports	Functional	Implemented
Responsive UI	Non-functional	Verified
API Latency < 500 ms	Non-functional	Verified
Cross-browser Compatibility	Non-functional	Verified
Password Hashing (BCrypt)	Security	Implemented
Token Refresh Mechanism	Security	Implemented

The authentication module correctly issued JSON Web Tokens upon successful login in all test cases. The token validation middleware rejected requests carrying expired tokens, tokens with invalid signatures, and requests with no Authorization header, returning HTTP 401 responses in each scenario. Password hashing was confirmed to store only BCrypt digest values in the Users table, with no plaintext credentials present in any database record inspected during testing.

Table III: Comparison with Commercial Alternatives

Feature	Proposed System	Mint (Free)	YNAB (Paid)
Cost	Free / Open-source	Free (ads)	USD 14.99/mo
Self-hostable	Yes	No	No
Expense Recording	Yes	Yes	Yes
Category Reports	Yes	Yes	Yes
Bank Integration	No (future)	Yes	Yes
Multi-currency	No (future)	Limited	Yes
Data Privacy Control	Full	Limited	Limited
Open-source	Yes	No	No

The transaction management module allowed users to create, read, update, and delete expense records reliably. Category-based filtering returned accurate result sets verified against direct database queries. Date-range queries on a test dataset of 5,000 transaction records executed within an average of 87 ms, well within the 500 ms non-functional requirement. The use of database indexes on the user_id and txn_date columns was instrumental in achieving this performance level.

The reporting module generated category-wise pie charts and monthly bar charts by consuming aggregation endpoints that performed MySQL GROUP BY operations server-side. Offloading aggregation to the database tier, rather than retrieving raw records and computing totals in Angular, substantially reduced both network payload size and client-side processing time. For a

user with 1,000 transactions, the aggregation endpoint returned a payload of approximately 280 bytes compared to an estimated 85 KB for the equivalent raw transaction data.

Responsiveness testing confirmed that the Angular application adapted its layout correctly to viewport widths ranging from 360 px (mobile) to 1920 px (large desktop monitor). The CSS Flexbox and Grid layout system, combined with Angular Material's responsive breakpoint utilities, ensured that form fields, data tables, and charts remained usable across all tested device categories. No horizontal scrollbars appeared on viewports wider than 360 px during testing.

A comparison with commercial alternatives (Table III) indicates that the proposed system achieves feature parity on core tracking and reporting functionality while eliminating subscription costs and dependency on third-party cloud infrastructure. The two capabilities absent from the current implementation—automatic bank integration and multi-currency support—are significant for users managing complex financial portfolios, but are non-essential for the target user segment of students and early-career professionals managing straightforward personal budgets. These capabilities are identified as priority items in the future work plan.

V. CONCLUSION

This paper presented the design, implementation, and evaluation of an Expense Tracker Web Application built on Angular 14, ASP.NET Core 6.0, and MySQL 8.0. The system provides a comprehensive yet approachable solution for personal financial management, supporting secure JWT-based authentication, category-based expense recording, flexible multi-parameter data filtering, and dynamic visual reporting through pie and bar charts. The three-tier architecture and RESTful API design ensure that the system is maintainable, extensible, and deployable on standard web hosting infrastructure without proprietary software dependencies.

The application successfully addresses the principal limitations identified in the review of existing tools: it is freely available, built entirely on open-source components, presents a simple and intuitive interface suitable for users without financial expertise, and affords complete data-privacy control through self-hosting. The multi-layer testing strategy validated both individual components and integrated system behaviour, confirming that all functional and non-functional requirements were satisfied by the delivered implementation.

The system does carry certain limitations that circumscribe its current applicability. Automatic synchronisation with bank accounts and payment instruments—a feature that significantly reduces manual data entry burden—is absent from the present version. Multi-currency support is similarly unavailable, limiting the system's utility for users who conduct financial activity in more than one currency. The reporting module, while sufficient for basic budget analysis, does not incorporate predictive analytics or anomaly detection capabilities that would proactively alert users to unusual spending patterns.

Future development work will address these limitations across three principal directions. First, integration with open banking APIs compliant with standards such as PSD2 (Europe) or Account Aggregator (India) will enable automatic transaction import, substantially reducing the manual effort required to maintain up-to-date financial records.

Second, a machine learning component will be incorporated to perform time-series anomaly detection on spending data, alerting users when expenditure in a particular category deviates significantly from established historical norms. Third, multi-currency support with real-time exchange-rate conversion will be implemented to broaden the system's applicability for internationally mobile users. Collectively, these enhancements will transform the current tool into a comprehensive personal financial management platform suitable for a broader and more demanding user base.

REFERENCES

1. R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 9th ed. New York, NY, USA: McGraw-Hill Education, 2019.
2. N. Bhatt and A. Mehta, "Mobile-based personal finance management: A study of user behaviour and application design," International Journal of Computer Applications, vol. 183, no. 6, pp. 12–18, Jun. 2021.
3. Google LLC, Angular Developer Guide, Version 14. Mountain View, CA, USA: Google Developers Documentation, 2022. [Online]. Available: https: //angular.io/docs
4. I. Sommerville, Software Engineering, 10th ed. London, UK: Pearson Education, 2016.
5. Microsoft Corporation, ASP.NET Core 6.0 Documentation: Building RESTful Web APIs. Redmond, WA, USA: Microsoft Developer Network, 2023. [Online]. Available: https ://docs.microsoft.com/en-us/aspnet/core
6. Oracle Corporation, MySQL 8.0 Reference Manual. Redwood City, CA, USA: Oracle Corporation, 2023. [Online]. Available: https: //dev.mysql.com/doc/refman/8.0/en
7. M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA, USA: Addison-Wesley Professional, 2002.
8. A. Silberschatz, H. F. Korth, and S. Sudarshan, Database System Concepts, 7th ed. New York, NY, USA: McGraw-Hill Education, 2019.
9. W. S. Vincent, Django for Beginners: Build Websites with Python and Django, 4th ed. Self-published, 2022. (Reference for REST API design patterns applied in this work.)
10. Reserve Bank of India, "Account Aggregator Framework," RBI Circular DNBR.PD.007/03.10.119/2016-17, Sep. 2016. [Online]. Available: https ://rbi.org.in