CCS 6440

Alexandros-Leonidas Tsirtiris

1.INTRODUCTION

This report chronicles the creation and deployment of an expense tracking application based on microservices to th e requirements of CCS6440 Cloud Engineering coursework. The project demonstrates re al-world application of current concepts in cloud engineering such as containerization, continuous integration/continuous deployment (CI/CD), and application monitoring.

The expense tracker is a system that keeps track of individual users' expenses in different categories through a RESTful API with a backend persistent database. It is an example illustrating the transition from traditional monolithic applications to cloud-native, distributed, scalable, and sustainable architectures.

2.System Architecture and Design Decisions

2.1 Architectural Overview

I chose a two-tier microservices architecture consisting of:

1. **Application Service Layer**: A Python-based RESTful API built with Flask framework.
2. **Data Persistence Layer**: MariaDB relational database for structured data storage.

This section complies with microservices principles by ensuring that every component may be independently created, deployed, and scaled and is accountable for only one function. A two-tier structure is chosen to mix simplicity with demonstrating key microservices principles required for this course.

2.2 Technology Stack Justification

Flask Framework: I selected Flask over alternatives like Django because of its lightweight nature and flexibility for building RESTful APIs. Flask's minimalist approach allows for precise control over the application structure without unnecessary overhead, making it ideal for microservices development.

MariaDB Database: The decision to use MariaDB instead of PostgreSQL or MySQL was driven by its excellent Docker support and strong compatibility with cloud environments. MariaDB provides robust ACID compliance essential for financial data integrity in an expense tracking application.

Docker Containerization: Containerization was implemented to ensure consistent deployment across different environments. This addresses the classic "it works on my machine" problem and provides isolation between services.

3.Database Design and Implementation

3.1 Schema Architecture

The database schema follows normalized design principles with three core entities:

- **Users Table**: Stores user credentials and account information with proper password hashing.

- **Expense Categories Table**: Maintains predefined expense categories (Food, Entertainment, Transportation, etc.).
- **Expenses Table**: Records individual expense transactions with foreign key relationships

This relational design ensures data integrity through referential constraints while maintaining flexibility for future enhancements. The use of auto-incrementing primary keys and timestamp fields provides audit capabilities and efficient indexing.

3.2 Data Integrity Measures

I implemented several data integrity measures:

- Foreign key constraints preventing orphaned records.
- Unique constraints on usernames and category names.
- Decimal precision for monetary values to avoid floating-point errors.
- Input validation at the application layer before database insertion.

4.API Development and Implementation

4.1 RESTful Design Principles

The API follows REST architectural constraints with clear resource-based URLs and appropriate HTTP methods:

- GET /api/categories - Retrieve all expense categories
- POST /api/users - Create new user accounts
- GET/POST/PUT/DELETE /api/expenses - Full CRUD operations for expenses
- GET /api/expenses/by_category/{id} - Category-specific expense filtering

Each endpoint returns consistent JSON responses with appropriate HTTP status codes. Error handling provides meaningful messages without exposing sensitive system information.

4.2 Business Logic Implementation

The application implements core business requirements:

- User registration with secure password hashing using Werkzeug.
- Expense categorization with predefined categories.
- Category-based expense filtering and reporting.
- Summary calculations with proper decimal handling for financial accuracy.

I chose to implement custom JSON encoding to handle Python Decimal objects and date serialization, ensuring consistent data representation across the API.

Containerization Strategy

5.1 Docker Implementation

The containerization approach uses multi-container deployment with Docker Compose:

Application Container: Built on Python 3.10 slim image to minimize attack surface while providing necessary runtime dependencies. The Dockerfile implements best practices including non-root user execution and layer optimization.

Database Container: Uses official MariaDB image with volume mounting for data persistence and initialization scripts for schema creation.

5.2 Orchestration Design

Docker Compose orchestrates the multi-container environment with:

- Dedicated network isolation between services.
- Health checks ensuring service availability before inter-service communication.
- Environment variable management for configuration.
- Volume mounting for persistent data storage.

This approach demonstrates infrastructure-as-code principles while maintaining development environment consistency.

6.Version Control and Git Workflow

6.1 Branching Strategy

I implemented a structured Git workflow with:

- **Main Branch**: Stable, production-ready code.
- **Development Branch**: Integration branch for new features.
- **Feature Branches**: Specific development tasks including containerization.

This branching strategy supports collaborative development while maintaining code quality through controlled integration points.

6.2 Commit History Management

All development changes are documented through meaningful commit messages following conventional commit format. The repository includes both application code and infrastructure

configuration, demonstrating GitOps principles where infrastructure changes are version-controlled alongside application code.

7. Continuous Integration and Deployment

7.1 CI/CD Pipeline Design

The GitHub Actions workflow implements automated testing and deployment:

Continuous Integration Phase:

- Automated Python dependency installation.
- Code quality checks using flake8 linting.
- Automated test execution on code changes.
- Multi-branch trigger support for both main and development branches.

Continuous Deployment Phase:

- Container image building and registry publishing.
- Environment-specific deployment configuration.
- Health verification post-deployment.

7.2 Testing Strategy

I implemented comprehensive testing including:

- Unit tests for individual API endpoints.
- Integration tests for database operations.
- End-to-end workflow testing covering complete user scenarios.
- Health check validation ensuring service availability.

The test suite covers critical business logic while maintaining fast execution times suitable for CI environments.

8.Monitoring and Observability

8.1 Application Metrics

The application exposes Prometheus-compatible metrics through a dedicated /metrics endpoint, providing visibility into:

- HTTP request rates and latency distribution

- Database operation performance metrics
- Application-specific business metrics
- System resource utilization

This monitoring approach follows observability best practices by providing quantitative insights into application behavior and performance characteristics.

8.2 Health Monitoring

A dedicated health check endpoint (/health) provides simple service availability verification used by both container orchestration and CI/CD pipelines. This endpoint performs basic connectivity tests without exposing sensitive system information.

9.Challenges and Solutions

9.1 Technical Challenges

Database Connection Management: Initial implementation experienced connection leaks under load testing. I resolved this by implementing SQLAlchemy connection pooling with proper session lifecycle management.

Container Networking: Early Docker networking configuration prevented inter-service communication. This was resolved through explicit network creation and service name resolution configuration.

CI/CD Pipeline Configuration: YAML syntax errors in GitHub Actions workflows required careful attention to indentation and structure. I addressed this by implementing workflow validation and using proven templates as starting points.

9.2 Development Process Challenges

Dependency Management: Managing Python dependencies across development and production environments required careful requirements.txt maintenance and virtual environment discipline.

Port Configuration: Local development port conflicts with system services required dynamic port mapping and environment-specific configuration management.

10.Security Considerations

Security measures implemented include:

- Password hashing using industry-standard algorithms

- Input validation preventing SQL injection attacks
- Environment variable management for sensitive configuration
- Container isolation limiting potential security exposure

These measures address common web application vulnerabilities while maintaining development productivity.

11.Future Enhancements and Scalability

The current architecture provides foundation for several enhancements:

- JWT-based authentication for stateless user sessions
- Horizontal scaling through container orchestration platforms
- Advanced analytics and reporting capabilities
- Mobile application integration through existing API endpoints

The microservices architecture supports these enhancements without requiring fundamental architectural changes.

12.Conclusion

This project effectively demonstrates the concepts of cloud engineering through real-world application of a microservices-based system. The expense tracker fulfills all provided requirements while showcasing modern development practices like containerization, automated testing, and continuous deployment.

The sample demonstrates how microservices architecture enables application components to be authored independently, for the purposes of scalability and maintainability demands of contemporary cloud applications. Integration with monitoring and observability offers operational insight necessary for production environments.

Development process underscored the need for appropriate tooling, automated tests, and infrastructure-as-code practices in managing complexity in distributed systems. These are directly applicable to enterprise-scale cloud engineering problems.

In doing so, I have obtained practical experience with the entire software development life cycle in the cloud, from the design phase all the way to production deployment and monitoring. The project is a good stepping stone for more complicated microservices deployments and demonstrates readiness for professional cloud engineering tasks.