Microservices Library Management System - Claude Code Prompts

Project Overview

Building a **true microservices-based** Library Management System with:

- **Multiple independent services** from the start
- **Local Windows VM development** with Docker
- **Cloud deployment ready** architecture
- **GitHub repo**: `navinprabhu/claude_libraryApp`
- OAuth JWT authentication with dedicated Auth Service
- API Gateway for service orchestration
- Service-to-service communication

Repository Structure Strategy

```
claude_libraryApp/
├── src/
│   ├── Services/
│   │   ├── LibraryApp.AuthService/
│   │   ├── LibraryApp.BookService/
│   │   ├── LibraryApp.MemberService/
│   │   └── LibraryApp.ApiGateway/
│   ├── Shared/
│   │   ├── LibraryApp.Shared.Models/
│   │   ├── LibraryApp.Shared.Infrastructure/
│   │   └── LibraryApp.Shared.Events/
│   └── Tests/
│       ├── LibraryApp.AuthService.Tests/
│       ├── LibraryApp.BookService.Tests/
│       ├── LibraryApp.MemberService.Tests/
│       └── LibraryApp.IntegrationTests/
├── docker/
│   ├── docker-compose.yml
│   ├── docker-compose.override.yml
│   └── Dockerfiles/
├── scripts/
│   ├── setup-local-dev.ps1
│   └── build-and-run.ps1
├── docs/
│   ├── api-documentation/
│   └── deployment-guides/
├── .github/
│   └── workflows/
│       ├── ci.yml
│       └── cd.yml
├── LibraryApp.sln
└── README.md
```

Modified Iterative Development Strategy

Phase 1: Repository Setup & Shared Infrastructure (Prompt 1)

Goal: Create GitHub repo structure and shared components

Prompt for Claude Code:

```
Set up a microservices solution structure for a Library Management System that will be stored in GitHub repo
"navinprabhu/claude_libraryApp":

1. Create the main solution file LibraryApp.sln

2. Create shared libraries:
   - LibraryApp.Shared.Models (DTOs, Enums, Constants)
   - LibraryApp.Shared.Infrastructure (Common interfaces, middleware, extensions)
   - LibraryApp.Shared.Events (Event models for service communication)

3. In LibraryApp.Shared.Models, create:
   - BookDto, CreateBookDto, UpdateBookDto
   - MemberDto, CreateMemberDto
   - BorrowingRecordDto, BorrowRequestDto
   - BookStatus enum (Available, Borrowed)
   - ApiResponse<T> wrapper class
   - PagedResult<T> for pagination

4. In LibraryApp.Shared.Infrastructure, create:
   - IRepository<T> interface
   - BaseRepository<T> implementation
   - IJwtTokenService interface
   - Common middleware (correlation ID, request logging)
   - Extension methods for service registration
   - Database configuration helpers

5. Add NuGet packages to shared projects:
   - Microsoft.EntityFrameworkCore.Abstractions
   - Microsoft.Extensions.DependencyInjection.Abstractions
   - Microsoft.AspNetCore.Http.Abstractions
   - System.ComponentModel.DataAnnotations

6. Create PowerShell scripts for Windows development:
   - setup-local-dev.ps1 (install Docker Desktop, clone repo, setup environment)
   - build-and-run.ps1 (build all services, start with Docker Compose)

Follow microservices patterns and prepare for multi-service architecture.
```

Phase 2: Authentication Service (Prompt 2)

Goal: Create dedicated authentication microservice

Prompt for Claude Code:

```
Create LibraryApp.AuthService as an independent microservice for authentication:

1. Create ASP.NET Core 8 Web API project: LibraryApp.AuthService

2. Project structure:
   LibraryApp.AuthService/
   ├── Controllers/
   │   └── AuthController.cs
   ├── Models/
   │   ├── LoginRequest.cs
   │   ├── LoginResponse.cs
   │   ├── TokenValidationRequest.cs
   │   └── User.cs
   ├── Services/
   │   ├── IAuthService.cs
   │   ├── AuthService.cs
   │   ├── IJwtTokenService.cs
   │   └── JwtTokenService.cs
   ├── Data/
   │   ├── AuthDbContext.cs
   │   └── UserRepository.cs
   ├── Configuration/
   │   └── JwtSettings.cs
   ├── Dockerfile
   ├── Program.cs
   └── appsettings.json

3. Implement JWT token generation and validation:
   - POST /api/auth/login (username/password -> JWT token)
   - POST /api/auth/validate (validate JWT token)
   - POST /api/auth/refresh (refresh expired tokens)
   - GET /api/auth/userinfo (get user claims from token)

4. Features:
   - EF Core with In-Memory database for users
   - JWT token generation with custom claims
   - Token validation middleware
   - Refresh token support
   - Password hashing (BCrypt)
   - Role-based claims (Admin, Member)

5. Docker configuration:
   - Create Dockerfile for containerization
   - Expose port 5001
   - Environment variable support
```

```
6. Add health checks endpoint: GET /health

7. Reference LibraryApp.Shared.Models and LibraryApp.Shared.Infrastructure

8. Seed default users:
   - admin/password (Admin role)
   - member1/password (Member role)

Create a production-ready authentication service that other microservices can use.
```

Phase 3: Book Service (Prompt 3)

Goal: Create book management microservice

Prompt for Claude Code:

Create LibraryApp.BookService as an independent microservice for book management:

1. Create ASP.NET Core 8 Web API project: LibraryApp.BookService

2. Project structure:
```
LibraryApp.BookService/
├── Controllers/
│   └── BooksController.cs
├── Models/
│   ├── Entities/
│   │   ├── Book.cs
│   │   └── BorrowingRecord.cs
│   └── Requests/
│       └── BorrowBookRequest.cs
├── Services/
│   ├── IBookService.cs
│   ├── BookService.cs
│   ├── IBorrowingService.cs
│   └── BorrowingService.cs
├── Data/
│   ├── BookDbContext.cs
│   ├── Repositories/
│   │   ├── IBookRepository.cs
│   │   ├── BookRepository.cs
│   │   ├── IBorrowingRecordRepository.cs
│   │   └── BorrowingRecordRepository.cs
│   └── DataSeeder.cs
├── Infrastructure/
│   ├── Middleware/
│   │   └── JwtAuthenticationMiddleware.cs
│   └── Extensions/
│       └── ServiceCollectionExtensions.cs
├── Dockerfile
├── Program.cs
└── appsettings.json
```

3. Implement RESTful endpoints:
- GET /api/books (get all books with pagination)
- GET /api/books/{id} (get book by ID)
- POST /api/books (create new book) [Admin only]
- PUT /api/books/{id} (update book) [Admin only]
- DELETE /api/books/{id} (delete book) [Admin only]
- POST /api/books/{id}/borrow (borrow book)
- POST /api/books/{id}/return (return book)
- GET /api/books/{id}/history (borrowing history)

```
4. Features:
   - EF Core with In-Memory database
   - JWT authentication middleware (validates tokens from AuthService)
   - Role-based authorization
   - AutoMapper for entity-DTO mapping
   - Custom exceptions with global exception handling
   - Structured logging with Serilog
   - Service-to-service communication preparation

5. Business logic:
   - Book availability checking
   - Borrowing validation (book available, member exists)
   - Return validation (book is actually borrowed)
   - Audit trail for all borrowing activities

6. Docker configuration:
   - Dockerfile for containerization
   - Expose port 5002
   - Environment variables for AuthService URL

7. Add health checks and readiness probes

8. Reference shared libraries and implement repository pattern

Ensure the service can validate JWT tokens issued by AuthService.
```

Phase 4: Member Service (Prompt 4)

Goal: Create member management microservice

Prompt for Claude Code:

Create LibraryApp.MemberService as an independent microservice for member management:

1. Create ASP.NET Core 8 Web API project: LibraryApp.MemberService

2. Project structure:
```
   LibraryApp.MemberService/
   ├── Controllers/
   │   └── MembersController.cs
   ├── Models/
   │   ├── Entities/
   │   │   └── Member.cs
   │   └── Requests/
   │       └── UpdateMemberRequest.cs
   ├── Services/
   │   ├── IMemberService.cs
   │   └── MemberService.cs
   ├── Data/
   │   ├── MemberDbContext.cs
   │   ├── Repositories/
   │   │   ├── IMemberRepository.cs
   │   │   └── MemberRepository.cs
   │   └── DataSeeder.cs
   ├── Infrastructure/
   │   ├── Middleware/
   │   │   └── JwtAuthenticationMiddleware.cs
   │   └── Extensions/
   │       └── ServiceCollectionExtensions.cs
   ├── Dockerfile
   ├── Program.cs
   └── appsettings.json
```

3. Implement RESTful endpoints:
   - GET /api/members (get all members) [Admin only]
   - GET /api/members/{id} (get member by ID)
   - POST /api/members (register new member)
   - PUT /api/members/{id} (update member)
   - DELETE /api/members/{id} (deactivate member) [Admin only]
   - GET /api/members/{id}/borrowed-books (get current borrowed books)
   - GET /api/members/{id}/borrowing-history (get borrowing history)

4. Features:
   - EF Core with In-Memory database
   - JWT authentication middleware
   - Role-based authorization (members can only access their own data)
   - Member profile management
   - Integration points for borrowing history (will call BookService)

```
   - Data validation and business rules

5. Business logic:
   - Member registration validation
   - Profile update restrictions
   - Member status management (Active, Suspended, Inactive)
   - Borrowing eligibility checks

6. Service-to-service communication:
   - HTTP client to call BookService for borrowing data
   - Circuit breaker pattern for resilience
   - Retry policies for failed calls

7. Docker configuration:
   - Dockerfile for containerization
   - Expose port 5003
   - Environment variables for other service URLs

8. Add health checks and external service dependency checks

9. Reference shared libraries and implement repository pattern

Prepare for inter-service communication with BookService for borrowing data.
```

Phase 5: API Gateway (Prompt 5)

Goal: Create API Gateway for service orchestration

Prompt for Claude Code:

```
Create LibraryApp.ApiGateway using Ocelot for microservices orchestration:

1. Create ASP.NET Core 8 project: LibraryApp.ApiGateway

2. Project structure:
  LibraryApp.ApiGateway/
  ├── Configuration/
  │   ├── ocelot.json
  │   ├── ocelot.Development.json
  │   └── ocelot.Production.json
  ├── Middleware/
  │   ├── CorrelationIdMiddleware.cs
  │   ├── RequestLoggingMiddleware.cs
  │   └── RateLimitingMiddleware.cs
  ├── Extensions/
  │   └── ServiceCollectionExtensions.cs
  ├── Dockerfile
  ├── Program.cs
  └── appsettings.json

3. Install and configure Ocelot:
  - Ocelot package for API Gateway functionality
  - Configure routes to all microservices
  - Load balancing configuration
  - Rate limiting and throttling

4. Route configuration in ocelot.json:
  - /api/auth/* -> AuthService (port 5001)
  - /api/books/* -> BookService (port 5002)
  - /api/members/* -> MemberService (port 5003)
  - Health checks aggregation

5. Features:
  - JWT authentication delegation to AuthService
  - Request/response transformation
  - CORS configuration for frontend clients
  - Request correlation ID generation
  - Centralized rate limiting
  - Request/response logging
  - Circuit breaker for downstream services

6. Gateway-specific functionality:
  - Aggregated health checks from all services
  - Service discovery preparation (for cloud deployment)
  - Request routing based on JWT claims
  - Global exception handling
```

```
7. Docker configuration:
   - Dockerfile for containerization
   - Expose port 5000 (main entry point)
   - Environment variables for service URLs

8. Security features:
   - JWT token validation before routing
   - Request sanitization
   - HTTPS enforcement preparation

9. Monitoring and logging:
   - Request tracing with correlation IDs
   - Performance metrics collection
   - Structured logging with Serilog

The API Gateway should be the single entry point for all client requests.
```

Phase 6: Docker Orchestration (Prompt 6)

Goal: Create Docker Compose for local development

Prompt for Claude Code:

```
Create Docker Compose configuration for local Windows VM development:

1. Create docker-compose.yml in project root:
   - AuthService container (port 5001)
   - BookService container (port 5002)
   - MemberService container (port 5003)
   - ApiGateway container (port 5000)
   - PostgreSQL database containers (one per service)
   - Redis for caching (optional)

2. Create docker-compose.override.yml for development:
   - Volume mounts for hot reload
   - Environment variables for development
   - Debug port exposures
   - Local network configuration

3. Service configuration:
   - Each service in its own container
   - Shared network for inter-service communication
   - Environment variable injection
   - Health checks for all services
   - Restart policies

4. Database configuration:
   - PostgreSQL containers for each service
   - Named volumes for data persistence
   - Environment variables for connection strings
   - Database initialization scripts

5. Networking:
   - Custom Docker network for service isolation
   - Service discovery via container names
   - Port mapping for external access

6. Create individual Dockerfiles for each service:
   - Multi-stage builds for optimization
   - .NET 8 runtime images
   - Health check instructions
   - Security best practices

7. Windows-specific considerations:
   - PowerShell scripts for container management
   - Windows container compatibility
   - Volume mount paths for Windows
   - File sharing configuration
```

```
8. Development workflow scripts:
   - build-all.ps1 (build all containers)
   - start-dev.ps1 (start development environment)
   - stop-all.ps1 (stop all containers)
   - logs.ps1 (view aggregated logs)
   - clean.ps1 (clean containers and volumes)

9. Environment configuration:
   - .env file for common variables
   - Service-specific environment files
   - JWT secret sharing between services
   - Database connection strings

Ensure easy local development setup on Windows VM with Docker Desktop.
```

Phase 7: Service Communication & Events (Prompt 7)

Goal: Implement inter-service communication

Prompt for Claude Code:

Implement service-to-service communication and event handling:

1. Update LibraryApp.Shared.Events with event models:
   - BookBorrowedEvent
   - BookReturnedEvent
   - MemberRegisteredEvent
   - MemberStatusChangedEvent

2. Create HTTP client communication:
   - In MemberService: HTTP client to call BookService
   - Service discovery using configuration
   - Polly for retry policies and circuit breakers
   - HttpClientFactory registration

3. Add event publishing (prepare for message queues):
   - IEventPublisher interface in shared infrastructure
   - In-memory event publisher for local development
   - Event serialization and correlation tracking

4. Update BookService:
   - Publish BookBorrowedEvent when book is borrowed
   - Publish BookReturnedEvent when book is returned
   - Add endpoint for MemberService to query borrowing status

5. Update MemberService:
   - Subscribe to book events for member history
   - HTTP calls to BookService for real-time data
   - Caching layer for frequently accessed data

6. Add correlation ID tracking:
   - Generate correlation ID in API Gateway
   - Pass through all service calls
   - Include in all log messages
   - Return in response headers

7. Error handling and resilience:
   - Timeout configurations for HTTP calls
   - Fallback strategies when services are unavailable
   - Graceful degradation of functionality
   - Circuit breaker patterns

8. Service health dependencies:
   - Health checks that include dependency checks
   - Readiness vs liveness probes
   - Cascading health status

```
9. Add integration tests:
   - Test service-to-service communication
   - Mock external service dependencies
   - End-to-end workflow testing

10. Monitoring service calls:
    - Log all inter-service communications
    - Track response times and failures
    - Alert on service communication issues

Ensure robust communication between microservices with proper error handling.
```

Phase 8: Testing Strategy (Prompt 8)

Goal: Comprehensive testing for microservices

Prompt for Claude Code:

```
Create comprehensive testing strategy for the microservices solution:

1. Unit Tests for each service:
   LibraryApp.AuthService.Tests/
   ├── Controllers/
   │   └── AuthControllerTests.cs
   ├── Services/
   │   ├── AuthServiceTests.cs
   │   └── JwtTokenServiceTests.cs
   └── TestHelpers/
       └── AuthTestData.cs

   Similar structure for BookService.Tests and MemberService.Tests

2. Integration Tests:
   LibraryApp.IntegrationTests/
   ├── ApiGatewayTests.cs
   ├── ServiceCommunicationTests.cs
   ├── EndToEndWorkflowTests.cs
   └── TestFixtures/
       ├── TestWebApplicationFactory.cs
       └── DatabaseFixture.cs

3. Testing tools and patterns:
   - xUnit for test framework
   - Moq for mocking dependencies
   - TestContainers for integration tests with real databases
   - Microsoft.AspNetCore.Mvc.Testing for API testing
   - FluentAssertions for readable assertions

4. Test categories:
   - Unit tests: Test individual components in isolation
   - Integration tests: Test service interactions
   - Contract tests: Verify API contracts between services
   - End-to-end tests: Test complete user workflows

5. Authentication testing:
   - Test JWT token generation and validation
   - Test role-based authorization
   - Test token expiration and refresh
   - Mock authentication for service tests

6. Service communication testing:
   - Test HTTP client calls between services
   - Test circuit breaker and retry policies
   - Test service unavailability scenarios
   - Mock external service dependencies

7. Database testing:
   - In-memory databases for unit tests
   - TestContainers with PostgreSQL for integration tests
   - Database seeding for consistent test data
   - Transaction rollback for test isolation

8. Docker testing:
   - Test container builds
   - Test service startup and health checks
   - Test inter-container communication
   - Test environment variable injection

9. Performance testing preparation:
   - Benchmark tests for critical endpoints
   - Load testing configuration
   - Memory and CPU usage testing

10. Continuous testing setup:
    - Test execution in Docker containers
    - Parallel test execution
    - Test result reporting
    - Code coverage analysis

Ensure high-quality, well-tested microservices with comprehensive test coverage.
```

Phase 9: CI/CD & GitHub Integration (Prompt 9)

Goal: Setup GitHub Actions and deployment pipeline

Prompt for Claude Code:

```
Create GitHub Actions workflows for CI/CD pipeline:

1. Create .github/workflows/ci.yml:
   - Trigger on push to main and pull requests
   - Build all microservices
   - Run unit and integration tests
   - Build and test Docker containers
   - Code coverage reporting
   - Security scanning

2. Create .github/workflows/cd.yml:
   - Trigger on releases
   - Build production Docker images
   - Push to container registry (Docker Hub or Azure ACR)
   - Deploy to staging environment
   - Run smoke tests
   - Deploy to production (manual approval)

3. Workflow features:
   - Matrix builds for different services
   - Caching for NuGet packages and Docker layers
   - Conditional deployments based on changed services
   - Environment-specific configurations
   - Secrets management for deployment credentials

4. Repository configuration:
   - Branch protection rules for main branch
   - Required status checks before merge
   - Pull request templates
   - Issue templates for bugs and features

5. Documentation:
   - README.md with setup instructions
   - API documentation generation
   - Architecture decision records (ADRs)
   - Deployment guides for different environments

6. Container registry setup:
   - Docker Hub repository configuration
   - Multi-architecture builds (amd64, arm64)
   - Image tagging strategy
   - Security scanning integration

7. Environment management:
   - Development environment (local Docker)
   - Staging environment (cloud-based)
   - Production environment (cloud-based)
   - Environment-specific configuration management

8. Monitoring and alerting setup:
   - Application Insights integration
   - Log aggregation configuration
   - Health check monitoring
   - Performance metrics collection

9. Security considerations:
   - Secrets management with GitHub Secrets
   - Dependency vulnerability scanning
   - Container image security scanning
   - Code quality checks

10. Documentation and processes:
    - Contributing guidelines
    - Code review processes
    - Release management procedures
    - Incident response procedures

Prepare the repository for professional development workflows and cloud deployment.
```

Phase 10: Cloud Deployment Preparation (Prompt 10)

Goal: Prepare for cloud deployment

Prompt for Claude Code:

Prepare the microservices solution for cloud deployment (Azure/AWS):

1. Infrastructure as Code:
   - Create ARM templates or Terraform scripts
   - Azure Container Instances or AWS ECS configuration
   - Application Gateway/Load Balancer setup
   - Database provisioning scripts
   - Key Vault/Secrets Manager integration

2. Cloud-specific configurations:
   - appsettings.Production.json for each service
   - Environment variable injection from cloud services
   - Connection string management
   - Logging integration with cloud providers

3. Service discovery and configuration:
   - Azure Service Discovery or AWS Service Discovery
   - Configuration management with Azure App Configuration
   - Feature flag integration
   - Dynamic configuration updates

4. Database migration:
   - EF Core migrations for production databases
   - Database seeding scripts for production
   - Backup and recovery procedures
   - Connection pooling optimization

5. Security enhancements:
   - HTTPS enforcement
   - API rate limiting
   - DDoS protection
   - Web Application Firewall configuration

6. Monitoring and observability:
   - Application Performance Monitoring setup
   - Distributed tracing configuration
   - Log aggregation and analysis
   - Health check monitoring

7. Scalability preparations:
   - Auto-scaling configurations
   - Load balancing strategies
   - Caching layers (Redis)
   - CDN integration for static content

8. Backup and disaster recovery:

```
   - Database backup strategies
   - Application data backup
   - Disaster recovery procedures
   - Multi-region deployment preparation

9. Performance optimizations:
   - Connection string optimizations
   - Memory and CPU limit configurations
   - Response compression
   - Caching strategies

10. Production readiness checklist:
    - Security review and penetration testing
    - Performance testing and optimization
    - Monitoring and alerting setup
    - Documentation and runbooks
    - Incident response procedures

Create a production-ready microservices solution ready for cloud deployment.
```

Windows VM Development Setup

Prerequisites Script (setup-prerequisites.ps1)

```powershell
# Enable WSL2 if not already enabled
# Install Docker Desktop for Windows
# Install Visual Studio 2022 or VS Code
# Install .NET 8 SDK
# Install Git for Windows
# Configure Git with GitHub credentials
```

Development Workflow

1. **Clone Repository**: `git clone https://github.com/navinprabhu/claude_libraryApp.git`
2. **Setup Environment**: Run `.\scripts\setup-local-dev.ps1`
3. **Build Services**: Run `.\scripts\build-and-run.ps1`
4. **Development**: Use Docker Compose for local development
5. **Testing**: Run tests in containers or locally
6. **Commit & Push**: Standard Git workflow with GitHub

Key Advantages of This Approach

True Microservices

- Independent services from day one
- Separate databases per service
- Clear service boundaries
- Independent deployment capability

Windows VM Optimized

- PowerShell scripts for Windows workflows
- Docker Desktop integration
- Windows-compatible file paths
- Local development optimizations

Cloud Ready

- Container-first architecture
- Environment-based configuration
- Infrastructure as Code preparation
- CI/CD pipeline integration

GitHub Integration

- Professional repository structure
- Automated workflows
- Documentation and processes
- Community-ready open source project

Recommendation for Execution

1. **Start with Phase 1-2** to establish foundation
2. **Test each service independently** before moving to next phase
3. **Use Docker from the beginning** to ensure consistency
4. **Commit frequently** to GitHub with meaningful messages
5. **Document as you go** for future reference

Would you like me to elaborate on any specific phase or adjust the strategy further based on your cloud provider preference (Azure vs AWS)?