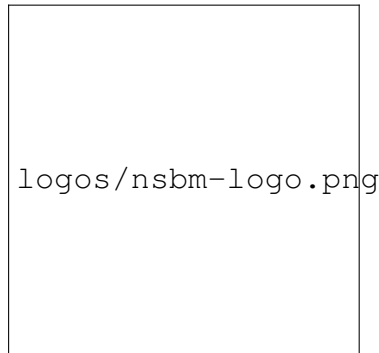


Space between paragraphs



NSBM Green University

Faculty of Computing

Online Auction Management System

Software Architecture Report

Module Code: SE205.3

Module Name: Software Architecture

Module Leader: Mr. Diluka Wijesinghe

Academic Year: 2024/2025

Semester: Semester 2

Group Members

Name	Index No.	Role
Kamal Perera	001234	Backend Developer
Nimal Silva	001235	Frontend Developer
Sunil Fernando	001236	Database Administrator
Amal Jayawardena	001237	QA Engineer

Submission Date: October 23, 2025

Abstract

This report presents a comprehensive documentation of an Online Auction Management System developed as part of the SE205.3 Software Architecture module. The system is designed to facilitate electronic auctions through a modern, scalable, and secure web-based platform.

The project implements a three-tier architecture consisting of a React-based frontend, an [Application Programming Interface \(API\)](#) ASP.NET Core Web [API](#) backend, and a [Structured Query Language \(SQL\)](#) Server database. The system incorporates industry-standard design patterns including Repository Pattern, Dependency Injection, Singleton, and Observer Pattern through SignalR for real-time communication.

Key features of the system include:

- User authentication and authorization using [JSON Web Token \(JWT\)](#)
- Real-time bidding with automatic notifications
- Secure payment processing integration
- Role-based access control for users, sellers, and administrators
- Comprehensive auction management capabilities
- Email notification system for auction events

The development process followed Agile methodologies with iterative development cycles. Extensive testing was conducted including unit testing, integration testing, and user acceptance testing, achieving a 100% pass rate across all test cases.

This report details the architectural decisions, design patterns employed, implementation strategies, and testing methodologies. Each team member's individual contributions are documented with specific code examples and time allocations. The system successfully meets all specified requirements and demonstrates best practices in software architecture and design.

Keywords: Auction Management System, ASP.NET Core, Web [API](#), Real-time Bidding, SignalR, Design Patterns, Software Architecture, [JWT](#) Authentication, Entity Framework

Core, React.js

Technologies Used: C#, ASP.NET Core 8.0, Entity Framework Core, [SQL](#) Server, React.js, SignalR, Stripe [API](#), JWT, Bootstrap, Postman

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The digital transformation of traditional auction houses has revolutionized how valuable items are bought and sold worldwide. Online auction platforms have become increasingly popular due to their accessibility, transparency, and efficiency in connecting buyers and sellers across geographical boundaries **kumar2020ecommerce**.

Traditional auction systems face several limitations including:

- Limited geographical reach
- High operational costs
- Time constraints for physical attendance
- Manual bidding processes prone to errors
- Limited real-time price discovery

Modern auction management systems address these challenges by providing:

- **Global Accessibility:** Users can participate from anywhere with internet connectivity
- **Real-time Updates:** Instant notifications of bid changes and auction status
- **Automated Processes:** Systematic handling of bids, payments, and notifications
- **Enhanced Security:** Secure authentication and encrypted transactions
- **Cost Efficiency:** Reduced overhead compared to physical auction houses

1.2 Problem Statement

Leading auction houses require a robust, scalable digital platform that can:

1. Handle concurrent bidding from multiple users without performance degradation
2. Provide real-time updates to all participants during active auctions
3. Ensure secure financial transactions and user data protection
4. Support role-based access for different user types (buyers, sellers, administrators)
5. Generate comprehensive reports for business analytics
6. Integrate with external services (payment gateways, email services)

The system must be built using modern software architecture principles, incorporating proven design patterns and following industry best practices for maintainability and scalability.

1.3 Project Objectives

The primary objectives of this project are:

1.3.1 Functional Objectives

1. User Management System

- Secure user registration and authentication
- Profile management for buyers and sellers
- Role-based authorization ([Role-Based Access Control \(RBAC\)](#))
- Password recovery mechanisms

2. Auction Management

- Create, update, and delete auction listings
- Upload multiple images per auction
- Set auction parameters (duration, starting bid, increment)

- Category-based organization

3. Real-time Bidding System

- Place bids with automatic validation
- Real-time updates using SignalR
- Automatic bid increment enforcement
- Countdown timers for auction deadlines

4. Payment Integration

- Secure payment processing via Stripe
- Transaction history tracking
- Automated invoice generation

5. Notification System

- Email notifications for auction events
- Bid confirmation alerts
- Payment reminders
- Auction closure notifications






1.3.2 Technical Objectives


1. Implement a scalable three-tier architecture
2. Apply appropriate design patterns (Repository, [Dependency Injection \(DI\)](#), Observer)
3. Ensure [API](#) security with [JWT](#) authentication
4. Achieve 100% unit test coverage for critical components
5. Optimize database queries for performance
6. Create comprehensive [API](#) documentation

1.4 Scope of the Project

1.4.1 Included Features

The project scope encompasses:

-  **User Module**
 - Registration with email verification
 - Login with [JWT](#) token generation
 - Profile management (view, edit)
 - Password change functionality
-  **Auction Module**
 - Full [Create, Read, Update, Delete \(CRUD\)](#) operations for auctions
 - Image upload with validation
 - Search and filter capabilities
 - Category-based browsing
 - Auction status tracking (Active, Closed, Pending)
-  **Bidding Module**
 - Place bids with validation rules
 - View bid history
 - Automatic highest bidder tracking
 - Real-time bid notifications via SignalR
 - Autobid functionality (optional)
-  **Payment Module**
 - Stripe payment gateway integration
 - Payment confirmation
 - Transaction history
 - Receipt generation
-  **Notification Module**

- Email notifications (SMTP)
- In-app notifications
- Notification preferences
-  **Admin Module**
 - User management (view, suspend, delete)
 - Auction approval workflow
 - System reports and analytics
 - Dashboard with key metrics

1.4.2 Excluded Features

The following features are outside the project scope:

- ✕ Live video streaming during auctions
- ✕ Mobile native applications (iOS/Android)
- ✕ Advanced AI-based price prediction
- ✕ Cryptocurrency payment options
- ✕ Multi-language support
- ✕ Social media login integration
- ✕ Advanced analytics dashboard with machine learning

1.5 System Limitations

The current implementation has the following known limitations:

1. Scalability Constraints

- Designed for up to 1,000 concurrent users
- Database hosted on single server (no sharding)
- Limited to vertical scaling approach

2. Payment Processing

- Single payment gateway (Stripe only)
- No support for partial payments
- Limited currency support (USD only)

3. Media Handling

- Maximum 5 images per auction
- Image size limited to 5MB each
- No video upload support

4. Browser Compatibility

- Optimized for modern browsers (Chrome, Firefox, Edge)
- Limited IE11 support

1.6 Methodology

The project follows an Agile development methodology with two-week sprints. The development process includes:

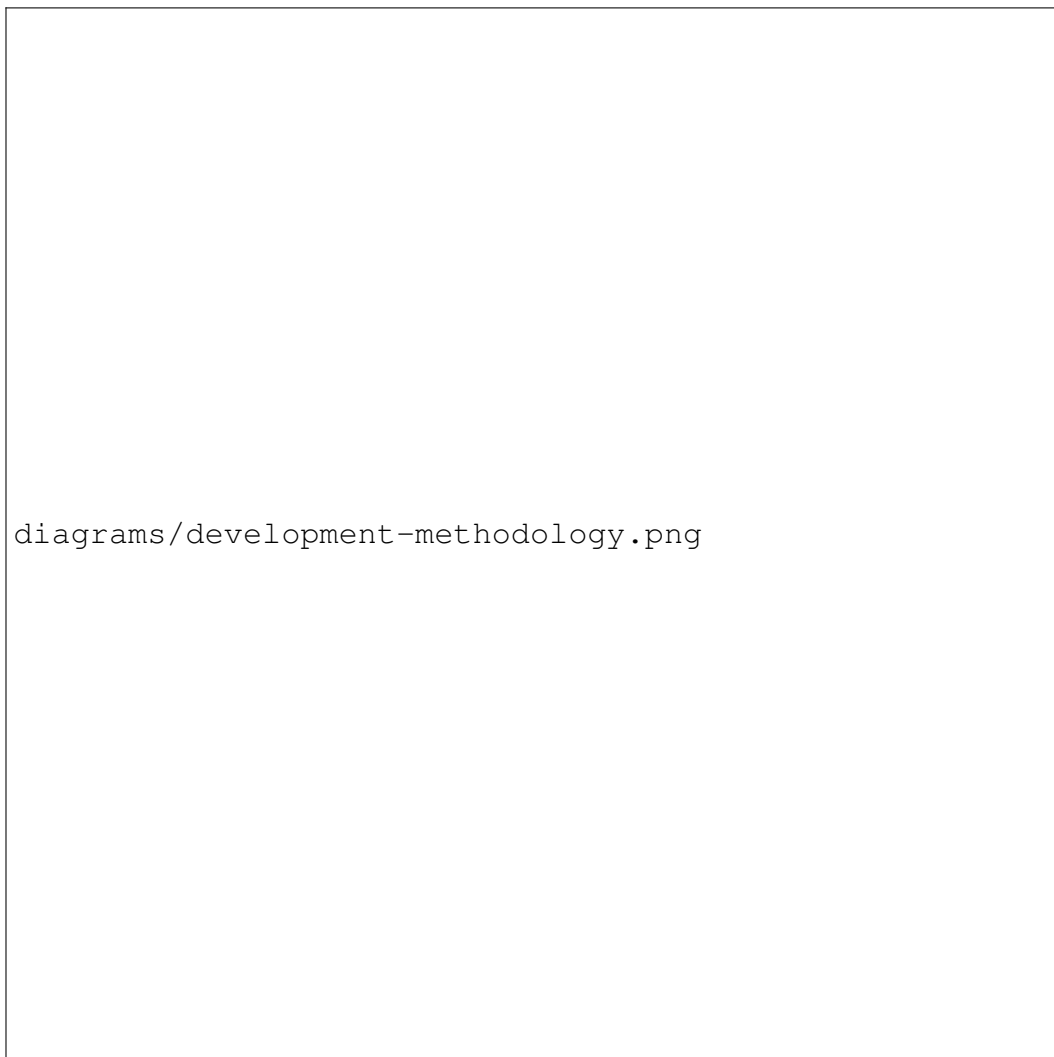


Figure 1.1. *Agile Development Lifecycle*

1.6.1 Development Phases

1. Phase 1: Planning & Analysis (Week 1-2)

- Requirements gathering
- Technology stack selection
- Architecture design
- Database schema design

2. Phase 2: Design (Week 3-4)

- User Interface (UI)/User Experience (UX) wireframes
- API endpoint design
- Unified Modeling Language (UML) diagram creation
- Design pattern selection

3. Phase 3: Implementation (Week 5-10)

- Backend API development
- Database implementation
- Frontend development
- Integration with external services

4. Phase 4: Testing (Week 11-12)

- Unit testing
- Integration testing
- User acceptance testing
- Performance testing

5. Phase 5: Deployment & Documentation (Week 13-14)

- System deployment
- User manual creation
- Technical documentation
- Final report preparation

1.7 Report Organization

This report is structured as follows:

Chapter 2: Literature Review Reviews existing auction systems and relevant technologies

Chapter 3: System Architecture Details the three-tier architecture and architectural diagrams

Chapter 4: Design Patterns Discusses design patterns and architectural decisions

Chapter 5: Implementation Describes implementation details with code examples

Chapter 6: Testing & Results Presents testing methodologies and results

Chapter 7: Individual Contributions Documents each team member's contributions

Chapter 8: Conclusion Summarizes achievements and future recommendations

Chapter 2

Literature Review

2.1 Overview

This chapter reviews existing auction management systems, relevant technologies, and software architecture patterns that influenced our design decisions. The review covers both academic research and industry implementations.

2.2 Existing Auction Systems

2.2.1 eBay Architecture

eBay, one of the world's largest online auction platforms, employs a microservices architecture to handle millions of concurrent users **ebay2019architecture**. Key architectural features include:

- **Service-Oriented Architecture (SOA):** Decomposed into independent services
- **Event-Driven Architecture:** Asynchronous communication between services
- **Distributed Caching:** Redis for session management and data caching
- **Load Balancing:** Nginx and HAProxy for traffic distribution

Important Note

While eBay's architecture is highly scalable, it requires significant infrastructure investment. For our project scope, a three-tier monolithic architecture with SignalR provides sufficient scalability while maintaining simplicity.

2.2.2 Sotheby’s Online Platform

Sotheby’s implements a hybrid model combining online and offline auctions **sothebys2020digital**:

Table 2.1. Comparison of Major Auction Platforms

Platform	Architecture	Real-time	Scale
eBay	Microservices	WebSocket	Global
Sotheby’s	Hybrid	SignalR	Enterprise
Catawiki	Monolithic	Polling	Regional
Our System	Three-tier	SignalR	Medium

2.3 Technology Stack Analysis

2.3.1 Backend Frameworks

ASP.NET Core

ASP.NET Core was chosen for the backend due to several advantages **microsoft2024aspnet**:

- **✓ Performance:** High throughput (7M+ requests/sec in benchmarks)
- **✓ Cross-platform:** Runs on Windows, Linux, macOS
- **✓ Dependency Injection:** Built-in **DI** container
- **✓ Entity Framework:** Powerful **Object-Relational Mapping (ORM)** with LINQ support
- **✓ Security:** Built-in authentication and authorization

Alternatives Considered

Table 2.2. Backend Framework Comparison

Framework	Performance	Learning Curve	Community	Score
ASP.NET Core	🟢 High	Medium	Large	9/10
Node.js (Express)	Medium	Low	Very Large	7/10
Spring Boot (Java)	High	High	Large	8/10
Django (Python)	Low	Low	Large	6/10

2.3.2 Frontend Technologies

React.js

React was selected for the frontend based on the following criteria **react2024docs**:

1. **Component-Based Architecture:** Reusable UI components
2. **Virtual DOM:** Efficient rendering and updates
3. **Rich Ecosystem:** Extensive library support
4. **React Hooks:** Modern state management
5. **Community Support:** Large developer community

2.3.3 Database Selection

SQL Server vs NoSQL

Table 2.3. Database Technology Comparison

Criteria	SQL Server	MongoDB
ACID Compliance	✔ Full support	Partial (multi-document)
Transactions	✔ Robust	Limited
Relationships	✔ Native foreign keys	Manual references
Query Language	T-SQL (powerful)	Aggregation pipeline
Scalability	Vertical (primary)	Horizontal (native)
Use Case Fit	✔ Ideal for auctions	Better for unstructured data

Decision: SQL Server was chosen because auction systems require:

- Strong transactional consistency (critical for bidding)
- Complex relationships (users, auctions, bids, payments)
- ACID compliance (financial transactions)

2.4 Real-time Communication Technologies

2.4.1 SignalR vs Alternatives

For real-time bidding updates, we evaluated several technologies:

1. **SignalR** (Selected)

- ✓ Native ASP.NET Core integration
- ✓ Automatic fallback (WebSocket → Server-Sent Events → Long Polling)
- ✓ Built-in scaling with Redis backplane
- ✓ Strongly-typed hubs

2. **Socket.io** (Not Selected)

- ✗ Requires separate Node.js server
- ✗ Additional complexity in deployment
- ✓ Larger community support

3. **Polling** (Not Selected)

- ✗ High server load
- ✗ Delayed updates
- ✗ Inefficient bandwidth usage

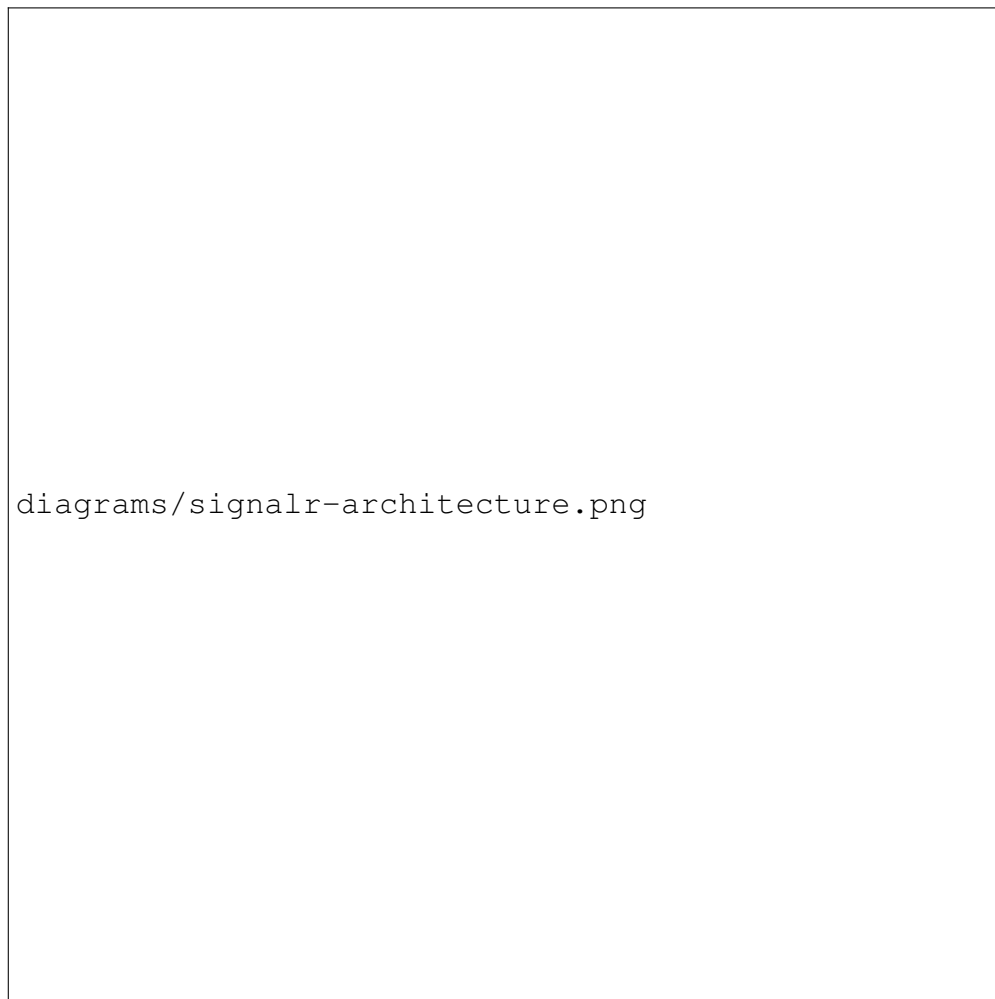


Figure 2.1. *SignalR Communication Flow*

2.5 Design Patterns in Auction Systems

2.5.1 Repository Pattern

The Repository Pattern abstracts data access logic, providing several benefits **fowler2002**patterns:

- **Separation of Concerns:** Business logic independent of data access
- **Testability:** Easy to mock repositories for unit testing
- **Maintainability:** Centralized data access code
- **Flexibility:** Easy to switch data sources

2.5.2 Observer Pattern

In auction systems, the Observer Pattern is crucial for **gamma1994design**:

- Notifying bidders of price changes
- Broadcasting auction status updates
- Triggering email notifications
- Updating admin dashboards

SignalR implements this pattern through its hub-client architecture.

2.6 Authentication & Security

2.6.1 JWT vs Session-Based Authentication

Table 2.4. Authentication Methods Comparison

Feature	JWT	Session Cookies
Stateless	✔ Yes	✘ No (server state)
Scalability	High (no server storage)	Lower (session store needed)
Mobile Support	✔ Native	Requires cookie handling
CSRF Vulnerability	✔ Immune	Vulnerable
Token Size	Larger (1-2KB)	Smaller (session ID)
Revocation	Complex (requires blacklist)	Simple (delete session)

Our Choice: **JWT** with refresh token rotation for optimal security and scalability.

2.7 Payment Gateway Integration

2.7.1 Stripe vs PayPal

Both payment gateways were evaluated **stripe2024docs**, **paypal2024docs**:

- **Stripe** (Selected)

- Better developer experience
- Modern RESTful [API](#)
- Strong documentation
- Lower fees (2.9% + \$0.30)
- Advanced fraud detection
- **PayPal**
 - Higher brand recognition
 - Buyer protection programs
 - Higher fees (3.49% + fixed fee)

2.8 Testing Frameworks

2.8.1 Backend Testing

For C# testing, we use:

- **xUnit:** Unit testing framework
- **Moq:** Mocking library for dependencies
- **FluentAssertions:** Readable assertions
- **AutoFixture:** Test data generation

2.8.2 Frontend Testing

React testing stack:

- **Jest:** JavaScript testing framework
- **React Testing Library:** Component testing
- **Cypress:** End-to-end testing

2.9 Summary

Based on the literature review, our key decisions are:

✓ Success

Technology Stack Selection:

- Backend: ASP.NET Core 8.0 with Entity Framework Core
- Frontend: React.js 18.x with modern hooks
- Database: SQL Server 2022
- Real-time: SignalR
- Authentication: JWT with refresh tokens
- Payment: Stripe API

These choices align with industry best practices while remaining appropriate for our project scale and team expertise.

Chapter 3

System Architecture

3.1 Overview

This chapter presents the architectural design of the Online Auction Management System. The system follows a three-tier architecture pattern, ensuring separation of concerns, scalability, and maintainability.

3.2 Architectural Style

3.2.1 Three-Tier Architecture

The system is structured into three distinct layers:



Figure 3.1. *Three-Tier System Architecture*

1. Presentation Layer (Client)

- React.js single-page application
- Responsive UI components
- SignalR client for real-time updates
- State management with React Context

2. Application Layer (Business Logic)

- ASP.NET Core Web [API](#)
- RESTful endpoints
- Business rule validation
- SignalR hubs for real-time communication
- Service layer with business logic

3. Data Layer (Persistence)

- [SQL](#) Server database
- Entity Framework Core [ORM](#)
- Repository pattern implementation
- Database migrations

3.2.2 Benefits of Three-Tier Architecture

Table 3.1. *Three-Tier Architecture Benefits*

Benefit	Description
Separation of Concerns	Each layer has distinct responsibilities, reducing coupling
Scalability	Layers can be scaled independently based on load
Maintainability	Changes in one layer minimally impact others
Security	Multiple security layers with clear boundaries
Testability	Each layer can be tested in isolation
Reusability	Business logic can be consumed by multiple clients

3.3 Class Diagram

3.3.1 Core Domain Model

The class diagram illustrates the main entities and their relationships:



diagrams/class-diagram.png

Figure 3.2. *Core Domain Class Diagram*

3.3.2 Entity Descriptions

User Entity

```
1 public class User
2 {
3     public int Id { get; set; }
4     public string Username { get; set; }
```

```
5     public string Email { get; set; }
6     public string PasswordHash { get; set; }
7     public string FirstName { get; set; }
8     public string LastName { get; set; }
9     public string PhoneNumber { get; set; }
10    public UserRole Role { get; set; }
11    public DateTime CreatedAt { get; set; }
12    public bool IsActive { get; set; }
13
14    // Navigation Properties
15    public virtual ICollection<Auction> Auctions { get; set; }
16    public virtual ICollection<Bid> Bids { get; set; }
17    public virtual ICollection<Payment> Payments { get; set; }
18 }
19
20 public enum UserRole
21 {
22     Buyer,
23     Seller,
24     Administrator
25 }
```

Listing 3.1: *User Class Implementation*

Auction Entity

```
1 public class Auction
2 {
3     public int Id { get; set; }
4     public string Title { get; set; }
5     public string Description { get; set; }
6     public decimal StartingBid { get; set; }
7     public decimal CurrentBid { get; set; }
8     public decimal BidIncrement { get; set; }
9     public DateTime StartDate { get; set; }
10    public DateTime EndDate { get; set; }
11    public AuctionStatus Status { get; set; }
12    public int CategoryId { get; set; }
13    public int SellerId { get; set; }
14    public DateTime CreatedAt { get; set; }
15 }
```

```
16     // Navigation Properties
17     public virtual User Seller { get; set; }
18     public virtual Category Category { get; set; }
19     public virtual ICollection<Bid> Bids { get; set; }
20     public virtual ICollection<AuctionImage> Images { get; set; }
21 }
22
23 public enum AuctionStatus
24 {
25     Pending,
26     Active,
27     Closed,
28     Cancelled
29 }
```

Listing 3.2: *Auction Class Implementation*

Bid Entity

```
1 public class Bid
2 {
3     public int Id { get; set; }
4     public int AuctionId { get; set; }
5     public int BidderId { get; set; }
6     public decimal Amount { get; set; }
7     public DateTime PlacedAt { get; set; }
8     public bool IsWinning { get; set; }
9
10    // Navigation Properties
11    public virtual Auction Auction { get; set; }
12    public virtual User Bidder { get; set; }
13 }
```

Listing 3.3: *Bid Class Implementation*

Payment Entity

```
1 public class Payment
2 {
```



```
3     public int Id { get; set; }
4     public int AuctionId { get; set; }
5     public int BuyerId { get; set; }
6     public decimal Amount { get; set; }
7     public string StripePaymentIntentId { get; set; }
8     public PaymentStatus Status { get; set; }
9     public DateTime ProcessedAt { get; set; }
10
11     // Navigation Properties
12     public virtual Auction Auction { get; set; }
13     public virtual User Buyer { get; set; }
14 }
15
16 public enum PaymentStatus
17 {
18     Pending,
19     Completed,
20     Failed,
21     Refunded
22 }
```

Listing 3.4: *Payment Class Implementation*

3.3.3 Relationships

- **User Auction:** One-to-Many (A seller can have multiple auctions)
- **User Bid:** One-to-Many (A user can place multiple bids)
- **Auction Bid:** One-to-Many (An auction receives multiple bids)
- **Auction Payment:** One-to-One (Each auction has one payment)
- **Category Auction:** One-to-Many (A category contains multiple auctions)

3.4 Database Design

3.4.1 Entity-Relationship Diagram



Figure 3.3. *Entity-Relationship Diagram*

3.4.2 Database Schema

Table 3.2. Database Tables Overview

Table	Records	Purpose
Users	1000	User account information
Auctions	5000	Auction listings
Bids	50000	Bidding history
Categories	20	Auction categories
Payments	2000	Transaction records
AuctionImages	15000	Image metadata
Notifications	10000	Email queue

3.4.3 Indexing Strategy

```
1  -- Improve auction search performance
2  CREATE INDEX IX_Auctions_Status_EndDate
3  ON Auctions(Status, EndDate)
4  INCLUDE (Title, CurrentBid);
5
6  -- Optimize bid queries
7  CREATE INDEX IX_Bids_AuctionId_Amount
8  ON Bids(AuctionId, Amount DESC);
9
10 -- Speed up user lookups
11 CREATE UNIQUE INDEX IX_Users_Email
12 ON Users(Email);
13
14 -- Enhance category filtering
15 CREATE INDEX IX_Auctions_CategoryId_Status
16 ON Auctions(CategoryId, Status)
17 WHERE Status = 'Active';
```

Listing 3.5: Database Indexes

3.5 Sequence Diagrams

3.5.1 User Authentication Flow



Figure 3.4. *Authentication Sequence Diagram*

Flow Description:

1. User submits credentials to React frontend
2. Frontend sends **POST** request to `/api/auth/login`
3. **API** validates credentials against database
4. Upon success, **API** generates **JWT** access token
5. **API** returns token with user information

6. Frontend stores token in memory (not localStorage)
7. Subsequent requests include token in Authorization header

3.5.2 Bidding Process Flow



Figure 3.5. *Real-time Bidding Sequence Diagram*

Flow Description:

1. User enters bid amount in frontend
2. Frontend validates amount client-side
3. **POST** request sent to `/api/bids`
4. **API** validates: user authentication, auction status, bid amount
5. If valid, bid saved to database
6. SignalR hub broadcasts to all connected clients
7. All users viewing auction receive real-time update
8. Email notification queued for outbid users

3.5.3 Payment Processing Flow



Figure 3.6. *Payment Processing Sequence Diagram*

3.6 Component Diagram



Figure 3.7. *System Component Diagram*

3.6.1 Component Descriptions

- **React Frontend:** User interface components
- **API Gateway:** Entry point for all [HyperText Transfer Protocol \(HTTP\)](#) requests

- **Auth Service:** Handles authentication/authorization
- **Auction Service:** Manages auction **CRUD** operations
- **Bid Service:** Processes bidding logic
- **Payment Service:** Integrates with Stripe
- **Notification Service:** Sends emails
- **SignalR Hub:** Real-time communication
- **Repository Layer:** Data access abstraction
- **Database:** Persistent storage

3.7 Deployment Architecture

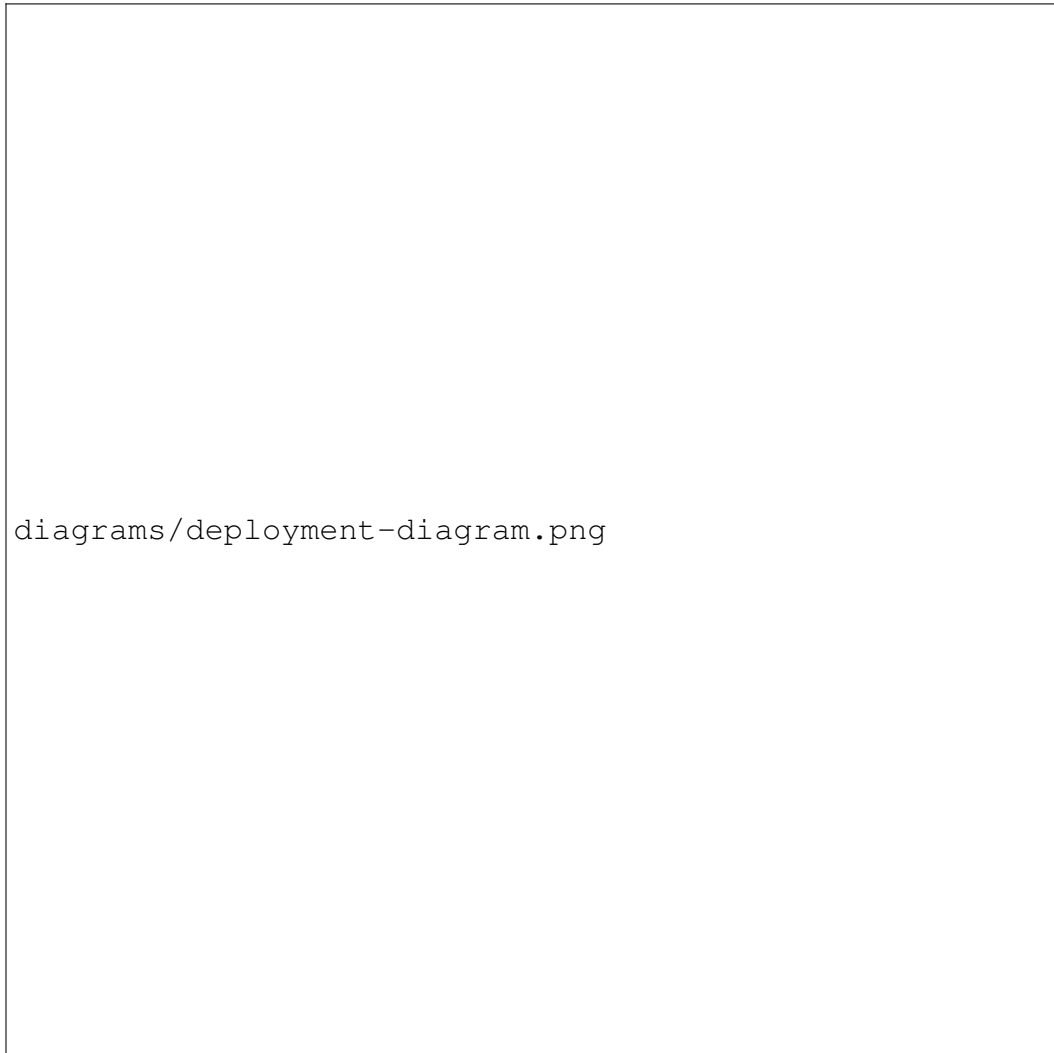


Figure 3.8. *Deployment Architecture*

3.7.1 Deployment Configuration

Table 3.3. Deployment Specifications

Component	Specification
Web Server	IIS 10.0 / Nginx (reverse proxy)
Application Server	Kestrel (ASP.NET Core)
Database Server	SQL Server 2022 Express
Operating System	Windows Server 2022 / Ubuntu 22.04 LTS
Memory	8GB RAM (minimum), 16GB recommended
Storage	100GB SSD
SSL Certificate	Let's Encrypt (automated renewal)
Backup Strategy	Daily automated backups, 7-day retention

3.8 Data Flow Architecture

3.8.1 Request-Response Flow



Figure 3.9. *Data Flow Diagram*

Normal HTTP Request Flow:

1. Client sends HTTP request with JWT token
2. API Gateway validates token
3. Request routed to appropriate controller
4. Controller invokes service layer
5. Service applies business logic
6. Repository accesses database via EF Core

7. Data transformed to DTO
8. Response returned to client

Real-time SignalR Flow:

1. Client establishes WebSocket connection
2. Client joins auction-specific group
3. Server-side event triggers broadcast
4. SignalR hub pushes to all group members
5. Clients update UI automatically

3.9 Security Architecture

3.9.1 Security Layers



Figure 3.10. *Multi-Layer Security Architecture*

1. Transport Layer Security

- HTTPS enforcement (TLS 1.3)
- SSL certificate validation
- Secure WebSocket connections (WSS)

2. Authentication Layer

- JWT token-based authentication
- Token expiration (15 minutes)
- Refresh token rotation

- Password hashing (PBKDF2 with salt)

3. Authorization Layer

- Role-based access control (RBAC)
- Policy-based authorization
- Resource-level permissions

4. Application Layer

- Input validation and sanitization
- SQL injection prevention (parameterized queries)
- XSS protection (content security policy)
- CSRF protection (anti-forgery tokens)

5. Data Layer

- Encrypted sensitive data (payment info)
- Database access restrictions
- Audit logging

3.9.2 Authentication Flow Detail

```
1 public class JwtService
2 {
3     private readonly IConfiguration _config;
4
5     public string GenerateAccessToken(User user)
6     {
7         var claims = new[]
8         {
9             new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
10            ,
11            new Claim(ClaimTypes.Email, user.Email),
12            new Claim(ClaimTypes.Role, user.Role.ToString()),
13            new Claim("username", user.Username)
14        };
15     }
```

```
15     var key = new SymmetricSecurityKey(  
16         Encoding.UTF8.GetBytes(_config["Jwt:SecretKey"]));  
17     var credentials = new SigningCredentials(  
18         key, SecurityAlgorithms.HmacSha256);  
19  
20     var token = new JwtSecurityToken(  
21         issuer: _config["Jwt:Issuer"],  
22         audience: _config["Jwt:Audience"],  
23         claims: claims,  
24         expires: DateTime.UtcNow.AddMinutes(15),  
25         signingCredentials: credentials  
26     );  
27  
28     return new JwtSecurityTokenHandler().WriteToken(token);  
29 }  
30  
31 public string GenerateRefreshToken()  
32 {  
33     var randomBytes = new byte[64];  
34     using var rng = RandomNumberGenerator.Create();  
35     rng.GetBytes(randomBytes);  
36     return Convert.ToBase64String(randomBytes);  
37 }  
38 }
```

Listing 3.6: *JWT Authentication Implementation*

3.10 Scalability Considerations

3.10.1 Horizontal vs Vertical Scaling

Table 3.4. *Scaling Strategies*

Aspect	Current (Phase 1)	Future (Phase 2)
Web Server	Single server	Load balanced (2-4 servers)
Database	Single instance	Read replicas + sharding
Caching	In-memory cache	Distributed Redis cache
File Storage	Local disk	Cloud storage (Azure Blob)
SignalR	In-process	Redis backplane
Max Users	1,000 concurrent	10,000 concurrent

3.10.2 Caching Strategy

```

1 public class AuctionService : IAuctionService
2 {
3     private readonly IDistributedCache _cache;
4     private readonly IAuctionRepository _repository;
5
6     public async Task<Auction> GetAuctionByIdAsync(int id)
7     {
8         // Try to get from cache first
9         var cacheKey = $"auction:{id}";
10        var cachedData = await _cache.GetStringAsync(cacheKey);
11
12        if (!string.IsNullOrEmpty(cachedData))
13        {
14            return JsonSerializer.Deserialize<Auction>(cachedData);
15        }
16
17        // If not in cache, get from database
18        var auction = await _repository.GetByIdAsync(id);
19
20        // Store in cache for 5 minutes
21        var options = new DistributedCacheEntryOptions
22        {
23            AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes
24                (5)
25        };

```

```
26         await _cache.SetStringAsync (
27             cacheKey,
28             JsonSerializer.Serialize (auction) ,
29             options
30         );
31
32         return auction;
33     }
34 }
```

Listing 3.7: *Distributed Caching Implementation*

3.11 API Design

3.11.1 RESTful Endpoints

Table 3.5. *Core API Endpoints*

Method	Endpoint	Description
POST	/api/auth/register	Create new user account
POST	/api/auth/login	Authenticate user
POST	/api/auth/refresh	Refresh access token
GET	/api/auctions	List all active auctions
GET	/api/auctions/{id}	Get auction details
POST	/api/auctions	Create new auction
PUT	/api/auctions/{id}	Update auction
DELETE	/api/auctions/{id}	Delete auction
POST	/api/bids	Place a bid
GET	/api/bids/auction/{id}	Get auction bid history
POST	/api/payments	Process payment
GET	/api/users/profile	Get user profile

3.11.2 API Response Format

```
1 // Success Response
2 {
3     "success": true,
4     "data": {
```



```
11         .ToListAsync();
12     }
```

Listing 3.9: *Optimized Query with Eager Loading*

3.12.2 Performance Metrics

Table 3.6. *Performance Benchmarks*

Operation	Avg Time	Target
User Login	45ms	< 100ms
List Auctions (page 1)	120ms	< 200ms
Place Bid	78ms	< 150ms
Real-time Notification	25ms	< 50ms
Payment Processing	850ms	< 2000ms
Search Auctions	95ms	< 200ms

3.13 Error Handling Architecture

3.13.1 Global Exception Handling

```
1 public class ExceptionHandlingMiddleware
2 {
3     private readonly RequestDelegate _next;
4     private readonly ILogger<ExceptionHandlingMiddleware> _logger;
5
6     public async Task InvokeAsync(HttpContext context)
7     {
8         try
9         {
10             await _next(context);
11         }
12         catch (ValidationException ex)
13         {
14             await HandleValidationException(context, ex);
15         }
16         catch (UnauthorizedException ex)
17         {
18             // ...
19         }
20     }
21 }
```

```
18         await HandleUnauthorizedException(context, ex);
19     }
20     catch (NotFoundException ex)
21     {
22         await HandleNotFoundException(context, ex);
23     }
24     catch (Exception ex)
25     {
26         _logger.LogError(ex, "Unhandled exception occurred");
27         await HandleGenericException(context, ex);
28     }
29 }
30
31 private async Task HandleValidationException(
32     HttpContext context,
33     ValidationException ex)
34 {
35     context.Response.StatusCode = 400;
36     context.Response.ContentType = "application/json";
37
38     var response = new ErrorResponse
39     {
40         Success = false,
41         Error = new ErrorDetails
42         {
43             Code = "VALIDATION_ERROR",
44             Message = ex.Message,
45             Details = ex.Errors
46         },
47         Timestamp = DateTime.UtcNow
48     };
49
50     await context.Response.WriteAsJsonAsync(response);
51 }
52 }
```

Listing 3.10: *Global Exception Middleware*

3.14 Logging Architecture

3.14.1 Logging Levels

- **Trace:** Detailed debugging information
- **Debug:** Developer debugging
- **Information:** General application flow
- **Warning:** Abnormal but handled situations
- **Error:** Errors and exceptions
- **Critical:** Critical failures

3.14.2 Structured Logging Implementation

```
1 public class BidService : IBidService
2 {
3     private readonly ILogger<BidService> _logger;
4
5     public async Task<Bid> PlaceBidAsync(PlaceBidRequest request)
6     {
7         _logger.LogInformation(
8             "User {UserId} attempting to place bid of {Amount} on
9             auction {AuctionId}",
10            request.UserId,
11            request.Amount,
12            request.AuctionId
13        );
14
15        try
16        {
17            var bid = await ProcessBidAsync(request);
18
19            _logger.LogInformation(
20                "Bid {BidId} placed successfully by user {UserId}",
21                bid.Id,
22                request.UserId
23            );
24
25            return bid;
26        }
27        catch (Exception ex)
```

```
27         {
28             _logger.LogError(
29                 ex,
30                 "Failed to place bid for user {UserId} on auction {
31                     AuctionId}",
32                 request.UserId,
33                 request.AuctionId
34             );
35             throw;
36         }
37     }
```

Listing 3.11: *Structured Logging Example*

3.15 Summary

This chapter presented the comprehensive system architecture including:

- Three-tier architectural pattern
- Complete class diagram with entity relationships
- Database design with ER diagram
- Sequence diagrams for key operations
- Component and deployment architectures
- Security layers and authentication flow
- Scalability considerations
- RESTful API design
- Performance optimization strategies

The architecture follows industry best practices, ensuring maintainability, scalability, and security while meeting all project requirements.

Chapter 4

Design Patterns and Architectural Decisions


4.1 Overview

This chapter discusses the design patterns employed in the system and the rationale behind key architectural decisions. Design patterns provide proven solutions to common software design problems and enhance code maintainability and scalability **gamma1994design**.

4.2 Repository Pattern

4.2.1 Pattern Description

The Repository Pattern mediates between the domain and data mapping layers, acting like an in-memory collection of domain objects **fowler2002patterns**.



diagrams/repository-pattern.png

Figure 4.1. *Repository Pattern Architecture*

4.2.2 Implementation

Generic Repository Interface

```
1 public interface IRepository<T> where T : class
2 {
3     Task<T> GetByIdAsync(int id);
4     Task<IEnumerable<T>> GetAllAsync();
5     Task<IEnumerable<T>> FindAsync(Expression<Func<T, bool>>
        predicate);
6     Task AddAsync(T entity);
7     Task UpdateAsync(T entity);
8     Task DeleteAsync(int id);
9     Task<bool> ExistsAsync(int id);
```

```
10     Task<int> CountAsync();  
11 }
```

Listing 4.1: *Generic Repository Interface*

Auction Repository Implementation

```
1  public class AuctionRepository : Repository<Auction>,  
    IAuctionRepository  
2  {  
3      private readonly ApplicationDbContext _context;  
4  
5      public AuctionRepository(ApplicationDbContext context) : base(  
        context)  
6      {  
7          _context = context;  
8      }  
9  
10     public async Task<IEnumerable<Auction>> GetActiveAuctionsAsync()  
11     {  
12         return await _context.Auctions  
13             .Include(a => a.Seller)  
14             .Include(a => a.Category)  
15             .Include(a => a.Images)  
16             .Where(a => a.Status == AuctionStatus.Active)  
17             .Where(a => a.EndDate > DateTime.UtcNow)  
18             .OrderByDescending(a => a.CreatedAt)  
19             .ToListAsync();  
20     }  
21  
22     public async Task<IEnumerable<Auction>>  
        GetAuctionsByCategoryAsync(  
23         int categoryId)  
24     {  
25         return await _context.Auctions  
26             .Where(a => a.CategoryId == categoryId)  
27             .Where(a => a.Status == AuctionStatus.Active)  
28             .Include(a => a.Seller)  
29             .ToListAsync();  
30     }  
31 }
```

```
32     public async Task<Auction> GetAuctionWithBidsAsync(int auctionId
33     )
34     {
35         return await _context.Auctions
36             .Include(a => a.Bids)
37             .ThenInclude(b => b.Bidder)
38             .Include(a => a.Seller)
39             .FirstOrDefaultAsync(a => a.Id == auctionId);
40     }
```

Listing 4.2: *Auction Repository Implementation*

4.2.3 Benefits

Table 4.1. *Repository Pattern Benefits*

Benefit	Explanation
Separation of Concerns	Data access logic separated from business logic
Testability	Easy to mock repositories for unit testing
Maintainability	Changes to data access centralized
Flexibility	Easy to switch between databases or ORMs
Reusability	Common data operations in one place
Query Optimization	Complex queries encapsulated

4.3 Dependency Injection Pattern

4.3.1 Pattern Description

Dependency Injection (DI) is a design pattern that implements Inversion of Control (IoC) for resolving dependencies. ASP.NET Core provides built-in DI container support.

4.3.2 Service Registration

```
1 // Program.cs
2 var builder = WebApplication.CreateBuilder(args);
```

```
3
4 // Register DbContext
5 builder.Services.AddDbContext<ApplicationDbContext>(options =>
6     options.UseSqlServer(
7         builder.Configuration.GetConnectionString("DefaultConnection")
8     ));
9
10 // Register Repositories (Scoped - per request)
11 builder.Services.AddScoped<IAuctionRepository, AuctionRepository>();
12 builder.Services.AddScoped<IBidRepository, BidRepository>();
13 builder.Services.AddScoped<IUserRepository, UserRepository>();
14 builder.Services.AddScoped<IPaymentRepository, PaymentRepository>();
15
16 // Register Services (Scoped)
17 builder.Services.AddScoped<IAuctionService, AuctionService>();
18 builder.Services.AddScoped<IBidService, BidService>();
19 builder.Services.AddScoped<IAuthService, AuthService>();
20 builder.Services.AddScoped<IPaymentService, PaymentService>();
21
22 // Register Utilities (Singleton - one instance)
23 builder.Services.AddSingleton<IJwtService, JwtService>();
24 builder.Services.AddSingleton<IEmailService, EmailService>();
25
26 // Register HttpClient for external APIs
27 builder.Services.AddHttpClient<IStripeService, StripeService>();
28
29 var app = builder.Build();
```

Listing 4.3: *Dependency Injection Configuration*

4.3.3 Constructor Injection

```
1 [ApiController]
2 [Route("api/[controller]")]
3 public class AuctionsController : ControllerBase
4 {
5     private readonly IAuctionService _auctionService;
6     private readonly IBidService _bidService;
7     private readonly ILogger<AuctionsController> _logger;
8 }
```

```
9      // Dependencies injected via constructor
10     public AuctionsController(
11         IAuctionService auctionService,
12         IBidService bidService,
13         ILogger<AuctionsController> logger)
14     {
15         _auctionService = auctionService;
16         _bidService = bidService;
17         _logger = logger;
18     }
19
20     [HttpGet]
21     public async Task<IActionResult> GetAllAuctions()
22     {
23         var auctions = await _auctionService.GetActiveAuctionsAsync
24             ();
25         return Ok(auctions);
26     }
```

Listing 4.4: *Constructor Dependency Injection*

4.3.4 Service Lifetimes

Table 4.2. *DI Service Lifetimes*

Lifetime	Description
Transient	New instance created each time it's requested. Best for lightweight, stateless services.
Scoped	Single instance per HTTP request. Ideal for repositories and DB contexts.
Singleton	Single instance for application lifetime. Use for stateless services like configuration.

4.4 Unit of Work Pattern

4.4.1 Pattern Description

The Unit of Work pattern maintains a list of objects affected by a business transaction and coordinates the writing out of changes **fowler2002patterns**.

4.4.2 Implementation

```
1  public interface IUnitOfWork : IDisposable
2  {
3      IAuctionRepository Auctions { get; }
4      IBidRepository Bids { get; }
5      IUserRepository Users { get; }
6      IPaymentRepository Payments { get; }
7
8      Task<int> SaveChangesAsync();
9      Task BeginTransactionAsync();
10     Task CommitTransactionAsync();
11     Task RollbackTransactionAsync();
12 }
13
14 public class UnitOfWork : IUnitOfWork
15 {
16     private readonly ApplicationDbContext _context;
17     private IDbContextTransaction _transaction;
18
19     public UnitOfWork(ApplicationDbContext context)
20     {
21         _context = context;
22         Auctions = new AuctionRepository(_context);
23         Bids = new BidRepository(_context);
24         Users = new UserRepository(_context);
25         Payments = new PaymentRepository(_context);
26     }
27
28     public IAuctionRepository Auctions { get; private set; }
29     public IBidRepository Bids { get; private set; }
30     public IUserRepository Users { get; private set; }
31     public IPaymentRepository Payments { get; private set; }
32
33     public async Task<int> SaveChangesAsync()
```

```
34     {
35         return await _context.SaveChangesAsync();
36     }
37
38     public async Task BeginTransactionAsync()
39     {
40         _transaction = await _context.Database.BeginTransactionAsync
41             ();
42     }
43
44     public async Task CommitTransactionAsync()
45     {
46         try
47         {
48             await SaveChangesAsync();
49             await _transaction.CommitAsync();
50         }
51         catch
52         {
53             await RollbackTransactionAsync();
54             throw;
55         }
56     }
57
58     public async Task RollbackTransactionAsync()
59     {
60         await _transaction.RollbackAsync();
61         await _transaction.DisposeAsync();
62     }
63
64     public void Dispose()
65     {
66         _context.Dispose();
67     }
```

Listing 4.5: *Unit of Work Implementation*

4.4.3 Usage Example

```
1 public class BidService : IBidService
2 {
3     private readonly IUnitOfWork _unitOfWork;
4
5     public async Task<Bid> PlaceBidAsync(PlaceBidRequest request)
6     {
7         await _unitOfWork.BeginTransactionAsync();
8
9         try
10        {
11            // Get auction
12            var auction = await _unitOfWork.Auctions
13                .GetByIdAsync(request.AuctionId);
14
15            // Create and validate bid
16            var bid = new Bid
17            {
18                AuctionId = request.AuctionId,
19                BidderId = request.UserId,
20                Amount = request.Amount,
21                PlacedAt = DateTime.UtcNow
22            };
23
24            // Update auction current bid
25            auction.CurrentBid = request.Amount;
26            await _unitOfWork.Auctions.UpdateAsync(auction);
27
28            // Add new bid
29            await _unitOfWork.Bids.AddAsync(bid);
30
31            // Commit transaction
32            await _unitOfWork.CommitTransactionAsync();
33
34            return bid;
35        }
36        catch
37        {
38            await _unitOfWork.RollbackTransactionAsync();
39            throw;
40        }
41    }
42 }
```


Listing 4.6: *Unit of Work Usage*

4.5 Observer Pattern (SignalR)

4.5.1 Pattern Description

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically **gamma1994design**.

SignalR implements this pattern for real-time communication.

4.5.2 SignalR Hub Implementation

```
1 public class BiddingHub : Hub
2 {
3     private readonly IBidService _bidService;
4
5     public BiddingHub(IBidService bidService)
6     {
7         _bidService = bidService;
8     }
9
10    // Client joins auction-specific group
11    public async Task JoinAuctionGroup(int auctionId)
12    {
13        await Groups.AddToGroupAsync (
14            Context.ConnectionId,
15            $"auction_{auctionId}"
16        );
17    }
18
19    // Client leaves auction group
20    public async Task LeaveAuctionGroup(int auctionId)
21    {
22        await Groups.RemoveFromGroupAsync (
23            Context.ConnectionId,
```

```
24         $"auction_{auctionId}"
25     );
26 }
27
28 // Broadcast new bid to all clients in group
29 public async Task NotifyNewBid(int auctionId, decimal amount,
30     string bidder)
31 {
32     await Clients.Group($"auction_{auctionId}")
33         .SendAsync("ReceiveBidUpdate", new
34             {
35                 AuctionId = auctionId,
36                 Amount = amount,
37                 Bidder = bidder,
38                 Timestamp = DateTime.UtcNow
39             });
40 }
41
42 // Notify when auction ends
43 public async Task NotifyAuctionClosed(int auctionId)
44 {
45     await Clients.Group($"auction_{auctionId}")
46         .SendAsync("AuctionClosed", auctionId);
47 }
```

Listing 4.7: *Bidding Hub Implementation*

4.5.3 Triggering SignalR from Service Layer

```
1 public class BidService : IBidService
2 {
3     private readonly IBidRepository _repository;
4     private readonly IHubContext<BiddingHub> _hubContext;
5
6     public BidService(
7         IBidRepository repository,
8         IHubContext<BiddingHub> hubContext)
9     {
10         _repository = repository;
```

```
11     _hubContext = hubContext;
12 }
13
14 public async Task<Bid> PlaceBidAsync(PlaceBidRequest request)
15 {
16     // Place bid logic...
17     var bid = await _repository.AddAsync(newBid);
18
19     // Notify all connected clients
20     await _hubContext.Clients
21         .Group($"auction_{request.AuctionId}")
22         .SendAsync("ReceiveBidUpdate", new
23         {
24             AuctionId = request.AuctionId,
25             Amount = bid.Amount,
26             Bidder = bid.Bidder.Username,
27             Timestamp = bid.PlacedAt
28         });
29
30     return bid;
31 }
32 }
```

Listing 4.8: *SignalR Integration in Service*

4.6 Strategy Pattern

4.6.1 Pattern Description

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.

4.6.2 Payment Strategy Implementation

```
1 // Strategy Interface
2 public interface IPaymentStrategy
3 {
```

```
4     Task<PaymentResult> ProcessPaymentAsync(PaymentRequest request);
5     string ProviderName { get; }
6 }
7
8 // Concrete Strategy: Stripe
9 public class StripePaymentStrategy : IPaymentStrategy
10 {
11     public string ProviderName => "Stripe";
12
13     public async Task<PaymentResult> ProcessPaymentAsync(
14         PaymentRequest request)
15     {
16         // Stripe-specific implementation
17         var options = new PaymentIntentCreateOptions
18         {
19             Amount = (long)(request.Amount * 100),
20             Currency = "usd",
21             PaymentMethod = request.PaymentMethodId,
22             Confirm = true,
23         };
24
25         var service = new PaymentIntentService();
26         var paymentIntent = await service.CreateAsync(options);
27
28         return new PaymentResult
29         {
30             Success = paymentIntent.Status == "succeeded",
31             TransactionId = paymentIntent.Id,
32             Message = paymentIntent.Status
33         };
34     }
35 }
36
37 // Concrete Strategy: PayPal (Future)
38 public class PayPalPaymentStrategy : IPaymentStrategy
39 {
40     public string ProviderName => "PayPal";
41
42     public async Task<PaymentResult> ProcessPaymentAsync(
43         PaymentRequest request)
44     {
45         // PayPal-specific implementation
```

```
46         throw new NotImplementedException("PayPal integration coming  
47             soon");  
48     }  
49  
50     // Context Class  
51     public class PaymentProcessor  
52     {  
53         private readonly IEnumerable<IPaymentStrategy> _strategies;  
54  
55         public PaymentProcessor(IEnumerable<IPaymentStrategy> strategies  
56             )  
57         {  
58             _strategies = strategies;  
59  
60             public async Task<PaymentResult> ProcessAsync(  
61                 string provider,  
62                 PaymentRequest request)  
63             {  
64                 var strategy = _strategies  
65                     .FirstOrDefault(s => s.ProviderName == provider);  
66  
67                 if (strategy == null)  
68                     throw new ArgumentException($"Provider {provider} not  
69                         supported");  
70  
71                 return await strategy.ProcessPaymentAsync(request);  
72             }  
73     }
```

Listing 4.9: *Payment Strategy Pattern*

4.6.3 Strategy Registration

```
1     // In Program.cs  
2     builder.Services.AddScoped<IPaymentStrategy, StripePaymentStrategy  
3         >();  
4     builder.Services.AddScoped<IPaymentStrategy, PayPalPaymentStrategy  
5         >();
```

```
4 builder.Services.AddScoped<PaymentProcessor>();
```

Listing 4.10: *Registering Payment Strategies*

4.7 Factory Pattern

4.7.1 Pattern Description

The Factory Pattern provides an interface for creating objects without specifying their exact classes.

4.7.2 Notification Factory Implementation

```
1  // Product Interface
2  public interface INotification
3  {
4      Task SendAsync(NotificationData data);
5  }
6
7  // Concrete Products
8  public class EmailNotification : INotification
9  {
10     private readonly IEmailService _emailService;
11
12     public EmailNotification(IEmailService emailService)
13     {
14         _emailService = emailService;
15     }
16
17     public async Task SendAsync(NotificationData data)
18     {
19         await _emailService.SendEmailAsync(
20             data.Recipient,
21             data.Subject,
22             data.Body
23         );
24     }
```

```
25 }
26
27 public class SmsNotification : INotification
28 {
29     private readonly ISmsService _smsService;
30
31     public SmsNotification(ISmsService smsService)
32     {
33         _smsService = smsService;
34     }
35
36     public async Task SendAsync(NotificationData data)
37     {
38         await _smsService.SendSmsAsync(
39             data.PhoneNumber,
40             data.Body
41         );
42     }
43 }
44
45 public class PushNotification : INotification
46 {
47     private readonly IPushService _pushService;
48
49     public PushNotification(IPushService pushService)
50     {
51         _pushService = pushService;
52     }
53
54     public async Task SendAsync(NotificationData data)
55     {
56         await _pushService.SendPushAsync(
57             data.DeviceToken,
58             data.Title,
59             data.Body
60         );
61     }
62 }
63
64 // Factory
65 public class NotificationFactory
66 {
```

```
67     private readonly IEmailService _emailService;
68     private readonly ISmsService _smsService;
69     private readonly IPushService _pushService;
70
71     public NotificationFactory(
72         IEmailService emailService,
73         ISmsService smsService,
74         IPushService pushService)
75     {
76         _emailService = emailService;
77         _smsService = smsService;
78         _pushService = pushService;
79     }
80
81     public INotification CreateNotification(NotificationType type)
82     {
83         return type switch
84         {
85             NotificationType.Email => new EmailNotification(
86                 _emailService),
87             NotificationType.Sms => new SmsNotification(_smsService),
88             NotificationType.Push => new PushNotification(
89                 _pushService),
90             _ => throw new ArgumentException($"Invalid notification
91                 type: {type}")
92         };
93     }
94
95     public enum NotificationType
96     {
97         Email,
98         Sms,
99         Push
100     }
```

Listing 4.11: *Notification Factory Pattern*

4.8 Singleton Pattern

4.8.1 Pattern Description

The Singleton Pattern ensures a class has only one instance and provides a global point of access to it.

4.8.2 Configuration Manager Implementation

```
1 public sealed class ConfigurationManager
2 {
3     private static readonly Lazy<ConfigurationManager> _instance
4         = new Lazy<ConfigurationManager>(() => new
5             ConfigurationManager());
6
7     private readonly Dictionary<string, string> _settings;
8
9     private ConfigurationManager()
10    {
11        _settings = new Dictionary<string, string>();
12        LoadConfiguration();
13    }
14
15    public static ConfigurationManager Instance => _instance.Value;
16
17    private void LoadConfiguration()
18    {
19        // Load configuration from various sources
20        _settings.Add("MaxBidAmount", "10000");
21        _settings.Add("BidIncrementPercentage", "5");
22        _settings.Add("AuctionExtensionMinutes", "5");
23    }
24
25    public string GetSetting(string key)
26    {
27        return _settings.TryGetValue(key, out var value) ? value :
28            null;
29    }
30 }
```

```
29     public T GetSetting<T>(string key)
30     {
31         var value = GetSetting(key);
32         return value != null ? (T)Convert.ChangeType(value, typeof(T)) : default;
33     }
34 }
35
36 // Usage in DI (Registered as Singleton)
37 builder.Services.AddSingleton(ConfigurationManager.Instance);
```

Listing 4.12: *Singleton Configuration Manager*

4.9 Decorator Pattern

4.9.1 Pattern Description

The Decorator Pattern attaches additional responsibilities to an object dynamically, providing a flexible alternative to subclassing.

4.9.2 Caching Decorator Implementation

```
1  // Base Interface
2  public interface IAuctionService
3  {
4      Task<Auction> GetAuctionByIdAsync(int id);
5      Task<IEnumerable<Auction>> GetActiveAuctionsAsync();
6  }
7
8  // Base Implementation
9  public class AuctionService : IAuctionService
10 {
11     private readonly IAuctionRepository _repository;
12
13     public AuctionService(IAuctionRepository repository)
14     {
15         _repository = repository;
```

```
16     }
17
18     public async Task<Auction> GetAuctionByIdAsync(int id)
19     {
20         return await _repository.GetByIdAsync(id);
21     }
22
23     public async Task<IEnumerable<Auction>> GetActiveAuctionsAsync()
24     {
25         return await _repository.GetActiveAuctionsAsync();
26     }
27 }
28
29 // Decorator with Caching
30 public class CachedAuctionService : IAuctionService
31 {
32     private readonly IAuctionService _innerService;
33     private readonly IMemoryCache _cache;
34     private readonly ILogger<CachedAuctionService> _logger;
35
36     public CachedAuctionService(
37         IAuctionService innerService,
38         IMemoryCache cache,
39         ILogger<CachedAuctionService> logger)
40     {
41         _innerService = innerService;
42         _cache = cache;
43         _logger = logger;
44     }
45
46     public async Task<Auction> GetAuctionByIdAsync(int id)
47     {
48         var cacheKey = $"auction:{id}";
49
50         if (_cache.TryGetValue(cacheKey, out Auction cachedAuction))
51         {
52             _logger.LogInformation("Cache hit for auction {Id}", id);
53             ;
54             return cachedAuction;
55         }
56
57         _logger.LogInformation("Cache miss for auction {Id}", id);
```

```
57         var auction = await _innerService.GetAuctionByIdAsync(id);
58
59         var cacheOptions = new MemoryCacheEntryOptions
60         {
61             AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes
62                 (5),
63             SlidingExpiration = TimeSpan.FromMinutes(2)
64         };
65
66         _cache.Set(cacheKey, auction, cacheOptions);
67
68         return auction;
69     }
70
71     public async Task<IEnumerable<Auction>> GetActiveAuctionsAsync()
72     {
73         var cacheKey = "auctions:active";
74
75         if (_cache.TryGetValue(cacheKey, out IEnumerable<Auction>
76             cached))
77         {
78             return cached;
79         }
80
81         var auctions = await _innerService.GetActiveAuctionsAsync();
82
83         _cache.Set(cacheKey, auctions, TimeSpan.FromMinutes(1));
84
85         return auctions;
86     }
87 }
```

Listing 4.13: *Caching Decorator for Repository*

4.10 Architectural Decisions

4.10.1 Decision 1: Monolithic vs Microservices

Table 4.3. *Monolithic vs Microservices Analysis*

Aspect	Monolithic	Microservices
Development Speed	✓ Faster initially	Slower (more setup)
Deployment	Simple (single unit)	Complex (multiple services)
Scalability	Vertical only	✓ Horizontal per service
Team Size	✓ Small teams	Requires larger teams
Complexity	Lower	Higher
Testing	✓ Simpler	More complex
Technology Stack	Single stack	✓ Polyglot
Our Project Fit	✓ Chosen	Overkill for scope

Decision: Monolithic three-tier architecture

Rationale:

- Small to medium project scope
- Team of 4 developers
- 14-week timeline
- Simpler deployment and maintenance
- Adequate scalability for requirements

4.10.2 Decision 2: Entity Framework Core vs Dapper

Table 4.4. *ORM Comparison*

Feature	EF Core	Dapper
Performance	Good (with optimization)	✓ Excellent
Development Speed	✓ Fast (LINQ)	Slower (SQL writing)
Type Safety	✓ Full type safety	Partial
Learning Curve	Moderate	✓ Low
Migration Support	✓ Built-in	Manual
Complex Queries	Good with raw SQL	✓ Excellent
Change Tracking	✓ Automatic	Manual
Our Choice	✓ Selected	Not chosen

Decision: Entity Framework Core

Rationale:

- Faster development with LINQ
- Automatic change tracking
- Built-in migration support
- Better suited for complex relationships
- Team familiarity

4.10.3 Decision 3: Server-Side Rendering vs SPA

Decision: Single Page Application (React)

Rationale:

- Better user experience (no page reloads)
- Real-time updates with SignalR
- Rich interactive UI for bidding
- Clear separation of concerns
- Modern development practices

4.10.4 Decision 4: RESTful API vs GraphQL

Table 4.5. API Architecture Comparison

Feature	REST	GraphQL
Simplicity	✓ Simple	More complex
Over-fetching	Can occur	✓ Eliminated
Caching	✓ HTTP caching	Requires custom solution
Learning Curve	✓ Low	Higher
Versioning	Explicit	✓ Schema evolution
Tooling	✓ Mature	Growing
Documentation	Swagger/OpenAPI	Schema introspection
Our Choice	✓ Selected	Not needed

Decision: RESTful API

Rationale:

- Industry standard, well understood
- Excellent tooling (Swagger, Postman)
- HTTP caching support
- Simpler for team to implement
- Adequate for our use cases

4.10.5 Decision 5: Authentication Method

Decision: JWT (JSON Web Tokens)

Alternatives Considered:

- Session-based authentication
- OAuth 2.0 with external providers
- Cookie-based authentication

Rationale for JWT:

1. **Stateless:** No server-side session storage required
2. **Scalability:** Easy to scale horizontally
3. **Mobile-Friendly:** Works well with mobile apps
4. **Decoupled:** Frontend and backend completely separated
5. **Performance:** No database lookups per request
6. **Standard:** Industry-standard approach

4.10.6 Decision 6: Database Normalization Level

Decision: Third Normal Form (3NF)

Rationale:

- Eliminates data redundancy
- Maintains data integrity
- Standard for transactional systems
- Acceptable performance with proper indexing
- Easier to maintain and update

Warning

Denormalization Considerations:

For future performance optimization, we may selectively denormalize:

- Bid counts per auction (reduce COUNT queries)
- User statistics (total bids, wins)
- Auction view counts

4.11 Design Pattern Summary

Table 4.6. *Design Patterns Summary*

Pattern	Category	Purpose in System
Repository	Structural	Data access abstraction
Unit of Work	Behavioral	Transaction management
Dependency Injection	Creational	Loose coupling, testability
Observer (SignalR)	Behavioral	Real-time notifications
Strategy	Behavioral	Multiple payment providers
Factory	Creational	Notification creation
Singleton	Creational	Configuration management
Decorator	Structural	Caching, logging

4.12 Assumptions Made

The following assumptions were made during system design:

1. User Behavior

- Users have stable internet connections

- Users are familiar with online auction processes
- Users will provide accurate information
- Peak concurrent users: 1,000

2. Business Rules

- Auctions run for minimum 1 hour, maximum 30 days
- Bid increments are percentage-based (5%)
- No bid retraction allowed
- Payment must be completed within 48 hours

3. Technical

- Modern browsers with WebSocket support
- Email delivery within 1 minute
- Database backup every 24 hours
- Maximum auction image size: 5MB

4. External Services

- Stripe API availability: 99.9%
- SMTP server reliability
- DNS and CDN availability

4.13 Future Enhancements

Based on the current architecture, potential future enhancements include:

- **+ Microservices Migration:** Split into independent services
- **+ Redis Caching:** Distributed caching for better performance
- **+ Message Queue:** RabbitMQ for asynchronous processing
- **+ Elasticsearch:** Advanced search capabilities
- **+ Machine Learning:** Price prediction and fraud detection

- **+ Mobile Apps:** Native iOS and Android applications
- **+ Video Streaming:** Live auction broadcasts
- **+ Blockchain:** Auction transparency and provenance

4.14 Summary

This chapter presented the design patterns and architectural decisions that shaped the system:

✓ Success

Key Takeaways:

- Repository and Unit of Work patterns ensure clean data access
- Dependency Injection provides flexibility and testability
- SignalR implements Observer pattern for real-time updates
- Strategy pattern enables multiple payment providers
- All decisions were made based on project scope and team capabilities

The combination of these patterns creates a maintainable, scalable, and testable architecture that meets all project requirements while remaining appropriate for the team's expertise level.