### **Freelance Project Board API - Comprehensive Technical Documentation**

Version: 1.0

Document Status: Final

Date: 12-06-2025

### **Table of Contents**

1. Introduction  
       \*   1.1. Project Vision  
       \*   1.2. Scope & Objectives  
       \*   1.3. Intended Audience
2. System Architecture & Design Philosophy  
       \*   2.1. N-Tier Architecture  
       \*   2.2. Core Design Patterns (Repository, Service, DI, DTO)  
       \*   2.3. Database Design & Conventions
3. Core Features & Implementation  
       \*   3.1. Authentication & User Lifecycle  
       \*   3.2. Role-Based Access Control (RBAC)  
       \*   3.3. Profile Management (Client & Freelancer)  
       \*   3.4. Project & Proposal Workflow  
       \*   3.5. File Management System  
       \*   3.6. Reputation System (Ratings)
4. Security Architecture In-Depth  
       \*   4.1. Authentication: JWT Access & Refresh Token Flow  
       \*   4.2. Authorization: Roles, Policies, and Record-Level Security  
       \*   4.3. Data Protection: Password Hashing & HTTPS/TLS Enforcement  
       \*   4.4. Input Validation & Threat Mitigation  
       \*   4.5. CORS Policy  
       \*   4.6. Rate Limiting
5. API Standards & Conventions  
       \*   5.1. Versioning Strategy  
       \*   5.2. Pagination Standard  
       \*   5.3. Error Handling & Response Structure
6. Observability: Logging, Monitoring & Tracing  
       \*   6.1. Structured Logging with Serilog  
       \*   6.2. Application Performance Monitoring (APM) with OpenTelemetry
7. Quality Assurance & Testing Strategy  
       \*   7.1. Unit Testing  
       \*   7.2. Executable Specification (Postman)  
       \*   7.3. Recommended Future Testing
8. Configuration & Deployment  
       \*   8.1. Environment Configuration (appsettings.json)  
       \*   8.2. Production Secret Management
9. **API Endpoint Reference**
10. **Conclusion & Future Roadmap**

### **1. Introduction**

#### **1.1. Project Vision**

To create a modern, secure, and highly scalable backend API that powers the Freelance Project Board platform. This API serves as the central nervous system, managing all data and business logic required to connect clients with skilled freelancers in an efficient and trustworthy digital marketplace.

#### **1.2. Scope & Objectives**

This project encompasses the design, development, and testing of a RESTful Web API. The scope includes the full lifecycle of all core platform activities: user registration, multi-faceted profile management, project creation and lifecycle tracking, a detailed proposal and bidding system, and a two-way rating system to build reputation.

#### **1.3. Intended Audience**

This document is intended for:

* **Backend Developers:** As the definitive guide for maintaining and extending the API.
* **Frontend Developers:** As a reference for API consumption, behavior, and data contracts.
* **DevOps Engineers:** For understanding configuration, deployment, and monitoring requirements.
* **Project Managers & Stakeholders:** As a detailed report on the project's features, architecture, and quality standards.

### **2. System Architecture & Design Philosophy**

The API is built on a foundation of modern software design principles to ensure it is maintainable, testable, and scalable.

#### **2.1. N-Tier Architecture**

A classic N-Tier architecture provides a strong separation of concerns:

* **Presentation Layer (Controllers):** Handles HTTP requests and responses. Responsible for routing, model binding, and returning appropriate status codes. It is deliberately kept "thin."
* **Business Logic Layer (Services):** Contains all business rules, validation logic, and orchestration of operations. This is the core of the application.
* **Data Access Layer (Repositories):** Manages all interactions with the database, abstracting the data source from the rest of the application.
* **Domain Layer (Models):** Defines the core entities of the application (e.g., User, Project).

#### **2.2. Core Design Patterns**

* **Repository Pattern:** Implemented to decouple the business logic (Service Layer) from the data access implementation (Entity Framework Core). A generic repository (IGenericRepository<T>) provides common CRUD operations, while specific repositories (IProjectRepository) handle complex, entity-specific queries.
* **Service Layer Pattern:** All business logic is encapsulated within service classes (IProjectService, IAuthService). This centralizes rules (e.g., a project must be 'Open' to be updated) and makes them easily testable.
* **Dependency Injection (DI):** ASP.NET Core's built-in DI container is used extensively to inject dependencies (like services and repositories) into controllers and other services. This promotes loose coupling and is fundamental to our unit testing strategy.
* **Data Transfer Objects (DTOs):** DTOs are used exclusively as the data contract for the API. This prevents exposing internal domain models, protects against mass-assignment vulnerabilities, and allows the API's public shape to evolve independently of the database schema.

#### **2.3. Database Design & Conventions**

* **Primary Keys:** All entities use GUID as their primary key type. This prevents key-guessing attacks and simplifies data synchronization in distributed or disconnected scenarios.
* **Soft Deletes:** All core entities inherit from a BaseEntity which includes an IsDeleted flag. DELETE operations set this flag to true instead of physically removing the row, preserving data history and referential integrity.
* **Audit Columns:** The BaseEntity also provides CreatedAt and UpdatedAt timestamps for every record, providing an out-of-the-box audit trail for data lifecycle events.

### **3. Core Features & Implementation**

#### **3.1. Authentication & User Lifecycle**

The AuthService manages the entire user lifecycle. User registration includes validation and password hashing using **BCrypt** (via ASP.NET Core Identity's default mechanisms). The login process validates credentials and issues a pair of JWTs (Access and Refresh tokens).

#### **3.2. Role-Based Access Control (RBAC)**

Authorization is a first-class citizen. [Authorize(Roles = "...")] attributes are used on controllers and actions to lock down functionality to specific user types (Admin, Client, Freelancer), ensuring the principle of least privilege.

#### **3.3. Profile Management (Client & Freelancer)**

The system supports two distinct profile types, managed by the ClientProfileService and FreelancerProfileService. This allows for tailored data models: ClientProfile focuses on company information, while FreelancerProfile focuses on individual skills, experience, and portfolio assets.

#### **3.4. Project & Proposal Workflow**

The ProjectService and ProposalService enforce a strict state machine:

1. A Client creates a Project, which enters the Open status.
2. A Freelancer submits a Proposal for an Open project. The proposal is Pending.
3. The Client can Accept a Proposal. This action triggers a state change in the corresponding Project, moving it to Assigned and linking the AssignedFreelancerId.
4. Once work is delivered, the Client marks the project as Completed. This unlocks the ability for both parties to leave a rating.

#### **3.5. File Management System**

The FileService provides centralized logic for handling file I/O. It accepts an IFormFile, stores it securely (e.g., on a local disk or cloud storage), and creates a corresponding metadata record in the database. This record links the file to its parent entity (e.g., a FreelancerProfile or Project) and stores information like filename, MIME type, and size.

#### **3.6. Reputation System (Ratings)**

The RatingService allows users to rate each other only in the context of a Completed project. It provides endpoints to calculate and retrieve a freelancer's average rating, which is a critical metric for their public reputation on the platform.

### **4. Security Architecture In-Depth**

#### **4.1. Authentication: JWT Access & Refresh Token Flow**

The API employs a stateless authentication model using JSON Web Tokens.

* **Access Token:** A short-lived (e.g., 60 minutes) token containing user claims (UserID, Role). It is sent in the Authorization header of every protected request.
* **Refresh Token:** A long-lived (e.g., 7 days), opaque token stored securely by the client. It is used exclusively at the /Auth/refresh-token endpoint to obtain a new access token without requiring the user to log in again.

#### **4.2. Authorization: Roles, Policies, and Record-Level Security**

Authorization is multi-layered:

* **Role-Based:** Controllers and endpoints are decorated with [Authorize(Roles = "Admin")] to restrict access based on the role claim in the JWT.
* **Record-Level:** Inside service methods, fine-grained checks are performed to ensure a user is the "owner" of a resource before allowing modification. For example, the UpdateProject service method verifies that the authenticated user's ID matches the ClientId of the project being updated.

#### **4.3. Data Protection: Password Hashing & HTTPS/TLS Enforcement**

* **Hashing:** Passwords are never stored. They are salted and hashed using a strong, one-way algorithm (BCrypt), making them computationally infeasible to reverse.
* **Transport Security:** The Kestrel web server is explicitly configured in Program.cs to enforce **TLS 1.2 or higher**. This encrypts all data in transit and protects against downgrade attacks.

#### **4.4. Input Validation & Threat Mitigation**

All DTOs that accept client input are validated using Data Annotations ([Required], [MaxLength], etc.). ModelState.IsValid is checked at the beginning of every POST and PUT action. This is the first line of defense against common vulnerabilities like Injection attacks and parameter tampering.

#### **4.5. CORS Policy**

A Cross-Origin Resource Sharing (CORS) policy is configured to only accept requests from a whitelist of trusted frontend origins, which are defined in appsettings.json. This prevents malicious websites from making requests to the API on behalf of a user.

#### **4.6. Rate Limiting**

To prevent abuse and ensure service availability, a per-user rate limiting policy is enforced. Using ASP.NET Core's PartitionedRateLimiter, each authenticated user is granted a configurable number of requests (e.g., 1000) per hour. Unauthenticated requests are limited by IP address.

### **5. API Standards & Conventions**

#### **5.1. Versioning Strategy**

The API is versioned via the URL path: /api/v{version}/.... This was implemented using the Asp.Versioning.Mvc library. This strategy allows for clear, explicit versioning and enables the introduction of breaking changes in future versions (e.g., v2) without affecting existing client integrations.

#### **5.2. Pagination Standard**

All list endpoints use a consistent pagination model.

* **Request:** ?PageNumber=1&PageSize=20
* Response: Wrapped in a PageResult<T> object containing the data and a pagination metadata block:  
      json     {       "data": [ ... ],       "pagination": {         "pageNumber": 1,         "pageSize": 20,         "totalPages": 15,         "totalRecords": 298       }     }

#### **5.3. Error Handling & Response Structure**

* **Global Middleware:** A custom ExceptionHandlingMiddleware catches all unhandled exceptions, logs them, and returns a generic 500 Internal Server Error response with a correlation ID.
* **Specific Errors:** Business logic errors are handled explicitly in controllers, returning appropriate 4xx status codes (400, 401, 403, 404, 409) with a clear JSON message body to aid client-side error handling.

### **6. Observability: Logging, Monitoring & Tracing**

#### **6.1. Structured Logging with Serilog**

Structured logging is implemented using Serilog.

* **Configuration:** Log levels and output sinks (Console, File) are configured in appsettings.json.
* **Format:** Logs are written in JSON format, which is easily parsable by log analysis tools.
* **Enrichment:** Every log entry is automatically enriched with contextual information. Most importantly, after authentication, the UserId is added to the logging context, allowing all actions performed by a specific user to be traced across all logs.

#### **6.2. Application Performance Monitoring (APM) with OpenTelemetry**

The API is instrumented with OpenTelemetry, the standard for application observability.

* **Instrumentation:** Automatic instrumentation is enabled for ASP.NET Core (incoming requests), HttpClient (outgoing requests), and Entity Framework Core.
* **Tracing:** This captures detailed performance data, including the exact SQL queries executed and their duration, allowing for easy identification of performance bottlenecks.
* **Exporting:** Traces are exported using the OTLP protocol to a compatible backend like Jaeger, enabling visualization and analysis of distributed traces.

### **7. Quality Assurance & Testing Strategy**

#### **7.1. Unit Testing**

The service layer is the focus of our unit testing strategy.

* **Frameworks:** We use **xUnit** as the test runner, **Moq** for creating mock implementations of repository and service dependencies, and **FluentAssertions** for its highly readable and expressive assertion syntax.
* **Approach:** Each public method in a service class has a corresponding set of tests that validate its primary success path as well as key failure paths (e.g., what happens when an entity is not found, or a business rule is violated). This ensures the core logic of the application is correct and reliable.

#### **7.2. Executable Specification (Postman)**

A comprehensive Postman Collection has been created, covering all 61 API endpoints. This collection is a critical project asset:

* **Automation:** It uses variables and scripts to automate the authentication flow and chain requests (e.g., creating a project and immediately using its ID to fetch it).
* **Demonstration:** It is the primary tool for conducting live, interactive showcases of the API's functionality.
* **Manual Testing:** It serves as an easy-to-use suite for manual regression testing before a new release.

#### **7.3. Future Testing**

* **Integration Testing:** The next logical step is to implement integration tests that use a real database (e.g., a test container running PostgreSQL). This would verify the correctness of repository logic, EF Core mappings, and the interaction between services and the database.
* **Contract Testing:** If the API will be consumed by critical, independently deployed clients, contract testing with a tool like Pact should be considered to prevent accidental breaking changes to the API contract.

### **8. Configuration & Deployment**

#### **8.1. Environment Configuration (appsettings.json)**

All environment-specific settings are defined in appsettings.json and can be overridden by appsettings.{Environment}.json. This includes database connection strings, JWT secrets, CORS origins, and logging configurations.

#### **8.2. Production Secret Management**

**CRITICAL:** For any production deployment, secrets **must not** be stored in source control. They must be managed through a secure mechanism such as:

* Azure Key Vault
* AWS Secrets Manager
* HashiCorp Vault
* Environment Variables injected by the hosting platform.

### **9. API Endpoint Reference**

A complete, static reference of all 61 endpoints is available in the accompanying **Excel/Markdown document**. It details the HTTP method, URI, description, authentication requirements, and all possible response codes for each endpoint.

However, the primary, interactive, and always-current source of documentation is the **Swagger/OpenAPI UI**, which is automatically generated from the code and can be accessed at the **/swagger** endpoint of the running application.

### **10. Conclusion & Future Roadmap**

This document outlines a feature-complete, secure, and production-ready v1.0 of the Freelance Project Board API. The architecture is robust, the features are comprehensive, and the standards for security, testing, and observability are high.

Recommended future enhancements include:

* **Audit Logging:** Implementing a dedicated audit trail for all data modifications to enhance security and compliance.
* **Real-time Notifications:** Integrating SignalR to push real-time updates to clients, creating a more dynamic and responsive user experience.