API Technical Specification Document

1. Overview

* API Type: RESTful API - default
* Protocol: HTTPS (TLS 1.2+)(optional) - default
* Content Type: application/json- Possible
* Authentication: JWT (Role-Based Authorization) - Possible
* Rate Limiting: 1000 requests/hour per user (configurable) - will work later
* Versioning: URI-based (/api/v1/) - will add additional features later at v2
* Documentation: Swagger/OpenAPI 3.0 - Possible

2. Functional Requirements

The API must support the following high-level features:

* User Authentication (Login, Logout, Token Refresh) - And also Register
* Role-Based Access Control
* CRUD operations on core entities
* File Upload support (e.g., profile picture, attachments) - Maybe an option to add
* Search, Filter, Sort, and Pagination
* Audit Logging of all data changes(Selection based on requirement)
* Optional: Real-time update notifications (SignalR/WebSockets)

3. Security Requirements

* JWT Bearer Authentication with Access + Refresh Tokens
* Password hashing using BCrypt
* Authorization based on user roles (e.g., Admin, Editor, Viewer)
* Input validation & sanitization
* HTTPS enforced across environments(Optional)
* CORS configuration for frontend clients
* Role-based policy enforcement in controllers(Based on requirement)

4. API Endpoints

4.1 Authentication

Method	Endpoint	Description
POST	/api/v1/auth/login	Authenticate user and return token
POST	/api/v1/auth/refresh	Refresh expired access token
POST	/api/v1/auth/logout	Invalidate current token
GET	/api/v1/auth/me	Get current user details

4.2 User Management

Method	Endpoint	Description
GET	/api/v1/users	List users (with pagination)
GET	/api/v1/users/{id}	Get user by ID
POST	/api/v1/users	Create new user
PUT	/api/v1/users/{id}	Update user details

DELETE /api/v1/users/{id} Delete user
4.3 Entity Management (Generic example)
Method Endpoint Description
GET /api/v1/items List items with filters
GET /api/v1/items/{id} Get item by ID
POST /api/v1/items Create new item
PUT /api/v1/items/{id} Update item
DELETE /api/v1/items/{id} Delete item
4.4 File Upload
Method Endpoint Description
POST /api/v1/files/upload Upload a file (multipart/form-data)
GET /api/v1/files/{filename} Download a file

5. Request & Response Standards
* Request Headers:
  * Authorization: Bearer <access_token>
  * Content-Type: application/json
* Standard Response Format:
json
CopyEdit
```
{
  "success": true,
  "message": "Item fetched successfully",
  "data": { /* object or array */ },
  "errors": null
}
```
* Error Response Format:
json
CopyEdit
```
{
  "success": false,
  "message": "Validation failed",
  "data": null,
  "errors": {
    "field": ["Error message"]
  }
}
```

6. Pagination Standard
* Supported via query parameters:
  * page (default: 1)
  * pageSize (default: 10, max: 100)

http
CopyEdit
GET /api/v1/items?page=2&pageSize=25
* Response example:
json
CopyEdit
```
{
  "data": [...],
  "pagination": {
    "totalRecords": 120,
    "page": 2,
    "pageSize": 25,
    "totalPages": 5
  }
}
```

7. Database Guidelines
* Use GUID or long as primary key
* Soft delete support using IsDeleted flag
* Audit columns: CreatedAt, CreatedBy, UpdatedAt, UpdatedBy

8. Logging & Monitoring
* Use structured logging (e.g., Serilog)
* Track all API calls with timestamps, user IDs, endpoint names
* Log errors with stack traces
* Enable Application Performance Monitoring (APM)

9. Testing
* Unit Testing for services and controllers (xUnit/NUnit)
* Integration Testing with in-memory DB
* Postman Collection for all endpoints
* Contract testing if third-party services are integrated

"Case Study 15: Freelance Project Board
Roles: Client, Freelancer
Features:
Post project requirements
Submit proposals
JWT authentication with role checks"

First focus on the phase 1 is to develop all the necessary functionalities starting with the models, theirs dbcontext using fluent API (will be using Postgres), then the Repositories (both interfaces and implementation), Automapper configuration, DTO's, service layer (both interfaces and implementation), controllers including search and pagination functionalities and exception handling throughout.

In the phase 2 of the project will be implementing extensive logging using serilog.

In the phase 3 will be doing extensive unit testing of the repositories and the services.

In the phase 4 will be implementing Authentication and Authorization (both role based and policy based) using jwt tokens securing all the necessary end points.

In the phase 5 will be implementing notifications using SIgnalR

In the phase 6 will implement the Rate limiting.

In the phase 7 will work on versioning by creating a v2 version with additional functions.

real world scenario and walkthrough of how it works so can model better like, a user can register both as a freelancer and a client, we need to decide separate fields for each while having a user model that links to both and will be helpful during authentication, and during registration what all we get from each, if there's any file upload involved, if in case of client what information we expect in real world, if it's a freelancer we would be additional getting about their skills, resume as attachment, expertise and related stuff, how we receive and store them will have a effect on our search functionality and same applies for the client too. As we had discussed about the rating too which will be based on the overall, what if it's new freelancer, we might also have a experience rating for client optionally rated by the freelancers, and after onboarding or login, what all a freelancer might expect? see all project postings from the client, ability to filter them, get extensive details about a specific project, submit a proposal to the client with necessary fields and attachments, view the status of the proposals, after the acceptance they might be able to work on the project and submit the project with files and stuffs related to that. While the client workflow would be like viewing all the freelancers, ability to create projects with all necessary details with optional attachments, view the proposals for the recruitment, accept or decline it, view the created projects, view the ongoing projects and the freelancer submissions and ability to rate the freelancer for completed project

RateLimiting, versioning, serilogging, documentation

Admin =

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1laWQiOiI4ZWM1ZGUxYi0xNDU3LTQyODEtOWI2My0zNDM1NTZjYzljMzgiLCJlbWFpbCI6ImFkbWluQGFwcC5jb20iLCJyb2xlIjoiQWRtaW4iLCJuYmYiOjE3NDk1NzMwNjgsImV4cCI6MTc0OTU3NjY2OCwiaWF0IjoxNzQ5NTczMDY4LCJpc3MiOiJodHRwczovL2xvY2FsaG9zdDo3MTk4IiwiYXVkIjoiaHR0cHM6Ly9sb2NhbGhvc3Q6NzE5OCJ9.6Sx3FmFUpUJ62Qrz4T4Ph_zygjZc2ojGELMFHvJCvkE

{
  "email": "admin@app.com",
  "password": "Test123"
}

{
  "email": "client@test.com",
  "password": "Test123"
}

{
  "email": "freelancer@test.com",
  "password": "Test123"
}

Client-1 =

```
.
├── Context
│   └── FreelanceContext.cs
├── Controllers
│   ├── AuthController.cs
│   ├── BaseApiController.cs
│   ├── ClientProfilesController.cs
│   ├── FilesController.cs
│   ├── FreelancerProfilesController.cs
│   ├── ProjectController.cs
│   ├── ProposalController.cs
│   ├── RatingsController.cs
│   ├── SkillsController.cs
│   └── UsersController.cs
├── DTOs
│   ├── Auth
│   │   ├── AuthResponseDto.cs
```

```
│   │   ├── ChangePasswordDto.cs
│   │   ├── RefreshTokenRequestDto.cs
│   │   ├── RevokeTokenRequestDto.cs
│   │   ├── UserLoginDto.cs
│   │   └── UserRegisterDto.cs
│   ├── ClientProfiles
│   │   ├── ClientProfileResponseDto.cs
│   │   ├── CreateClientProfileDto.cs
│   │   └── UpdateClientProfileDto.cs
│   ├── Files
│   │   └── FileResponseDto.cs
│   ├── FreelancerProfiles
│   │   ├── CreateFreelancerProfileDto.cs
│   │   ├── FreelancerProfileResponseDto.cs
│   │   └── UpdateFreelancerProfileDto.cs
│   ├── Projects
│   │   ├── CreateProjectDto.cs
│   │   ├── ProjectListDto.cs
│   │   ├── ProjectResponseDto.cs
│   │   └── UpdateProjectDto.cs
│   ├── Proposals
│   │   ├── CreateProposalDto.cs
│   │   ├── ProposalResponseDto.cs
│   │   └── UpdateProposalStatusDto.cs
│   ├── Ratings
│   │   ├── CreateRatingDto.cs
│   │   └── RatingResponseDto.cs
│   ├── Skills
│   │   ├── CreateSkillDto.cs
│   │   └── SkillDto.cs
│   └── Users
│       ├── UpdateUserDto.cs
│       └── UserResponseDto.cs
├── ErrorHandling
│   ├── ErrorDetails.cs
│   └── ExceptionHandlingMiddleware.cs
├── Helpers
│   ├── ClientFilter.cs
│   ├── FreelancerFilter.cs
│   ├── PageResult.cs
│   ├── PaginationParams.cs
│   ├── ProjectFilter.cs
```

```
│   ├── SkillFilter.cs
│   └── UserFilter.cs
├── Interfaces
│   ├── Repositories
│   │   ├── IClientProfileRepository.cs
│   │   ├── IFileRepository.cs
│   │   ├── IFreelancerProfileRepository.cs
│   │   ├── IFreelancerSkillRepository.cs
│   │   ├── IGenericRepository.cs
│   │   ├── IProjectRepository.cs
│   │   ├── IProjectSkillRepository.cs
│   │   ├── IProposalRepository.cs
│   │   ├── IRatingRepository.cs
│   │   ├── ISkillRepository.cs
│   │   └── IUserRepository.cs
│   └── Services
│       ├── IAuthService.cs
│       ├── IClientProfileService.cs
│       ├── IFileService.cs
│       ├── IFreelancerService.cs
│       ├── IProjectService.cs
│       ├── IProposalService.cs
│       ├── IRatingService.cs
│       ├── ISkillService.cs
│       └── IUserService.cs
├── Mapper
│   └── MappingProfile.cs
├── Models
│   ├── BaseEntity.cs
│   ├── ClientProfile.cs
│   ├── File.cs
│   ├── FreelancerProfile.cs
│   ├── FreelancerSkill.cs
│   ├── Project.cs
│   ├── ProjectSkill.cs
│   ├── Proposal.cs
│   ├── Rating.cs
│   ├── Skill.cs
│   └── User.cs
├── Program.cs
├── Properties
│   └── launchSettings.json
```

```
├── Repositories
│   ├── ClientProfileRepository.cs
│   ├── FileRepository.cs
│   ├── FreelancerProfileRepository.cs
│   ├── FreelancerSkillRepository.cs
│   ├── GenericRepository.cs
│   ├── ProjectRepository.cs
│   ├── ProjectSkillRepository.cs
│   ├── ProposalRepository.cs
│   ├── RatingRepository.cs
│   ├── SkillRepository.cs
│   └── UserRepository.cs
├── Services
│   ├── AuthService.cs
│   ├── ClientProfileService.cs
│   ├── FileService.cs
│   ├── FreelancerProfileService.cs
│   ├── ProjectService.cs
│   ├── ProposalService.cs
│   ├── RatingService.cs
│   ├── SkillService.cs
│   └── UserService.cs
├── appsettings.Development.json
├── appsettings.json
```

, Versioning: URI-based (/api/v1/)

Preparation:

- Postman Collection is fully built out with a folder for each controller.
- The `{{baseUrl}}`, `{{jwt_token}}` variables are in place.
- We'll add more variables as we go: `clientId`, `freelancerId`, `clientProfileId`, `freelancerProfileId`, `projectId`, `proposalId`, `skillId`, `ratingId`, `attachmentId`.
- Have at least three pre-registered users ready in your database for the demo: `client@example.com`, `freelancer@example.com`, and `admin@example.com`.

The Ultimate Showcase: A 61-Endpoint Grand Tour

Operator: The person running the Postman demo.
Narrator: The person explaining what's happening.

(The demo begins)

Narrator: "Welcome to the complete demonstration of the Freelance Project Board API. We will now walk through every single one of its 61 endpoints to showcase its full capabilities, from authentication and user management to the detailed project lifecycle. We'll be using Postman to interact with the API live."

Act I: The Foundation - Auth & User Management

Narrator: "Every platform begins with its users. Let's start by registering our key personas and then see how an Administrator can manage them."

Operator Actions:

1. **`Auth` Folder:**
    - Run **`POST /Auth/register`** with `client@example.com` details.
    - Run **`POST /Auth/register`** with `freelancer@example.com` details.
    - Run **`POST /Auth/login`** as `admin@example.com`. **(The `jwt_token` is now the Admin's)**.
        - **Narrator:** "We've registered a Client and a Freelancer, and now we've logged in as an Administrator."
    -
2.
3. **`Users` Folder:**
    - Run **`GET /Users`**.

        - **Narrator:** "As an Admin, we have the privilege to view a paginated list of all users on the platform."
    -
    - Run **`GET /Users/{id}`** using the ID of the `client@example.com` user from the previous response. **(Save this ID as `clientId`)**.
        - **Narrator:** "The Admin can fetch the detailed profile of any specific user."
    -
    - Run **`GET /Users/{id}`** using the ID of the `freelancer@example.com` user. **(Save this ID as `freelancerId`)**.
    - **Login as Client:** Run **`POST /Auth/login`** with `client@example.com`.
    - Run **`PUT /Users/{id}`** using the `{{clientId}}` and update their name.
        - **Narrator:** "Now logged in as the client, they can update their own personal details."
    -
    - Run **`PUT /Users/{id}/change-password`**.
        - **Narrator:** "And they have the ability to change their own password securely."
    -
4.

Act II: Building Identities - Profiles & Skills

Narrator: "With user accounts established, our members need to build their professional identities on the platform."

Operator Actions:

1. **`Skills` Folder (As Admin):**
    - Run **`POST /Auth/login`** as `admin@example.com`.
    - Run **`POST /Skills`** to create a new skill like "C#".
        - **Narrator:** "The Admin is responsible for curating the list of available skills."
    -
    - Run **`GET /Skills`**. In the response, find the ID of "C#" and save it as `skillId`.
        - **Narrator:** "We can retrieve a full, paginated list of all skills."
    -
    - Run **`GET /Skills/{id}`** using `{{skillId}}`.
        - **Narrator:** "And we can fetch a specific skill by its ID."
    -

2.
3. **`Client Profiles` Folder (As Client):**
    - Run **`POST /Auth/login`** as `client@example.com`.
    - Run **`POST /ClientProfiles`**. Save the returned profile `id` as `clientProfileId`.
        - **Narrator:** "The client, now authenticated, creates their company profile."
    -
    - Run **`GET /ClientProfiles/me`**.
        - **Narrator:** "They can easily retrieve their own profile using a dedicated endpoint."
    -
    - Run **`PUT /ClientProfiles/{id}`** using `{{clientProfileId}}` to update the company description.
        - **Narrator:** "And they can update it as needed."
    -
4.
5. **`Freelancer Profiles` Folder (As Freelancer):**
    - Run **`POST /Auth/login`** as `freelancer@example.com`.
    - Run **`POST /FreelancerProfiles`** including the `{{skillId}}` in the skills array. Save the returned `id` as `freelancerProfileId`.
        - **Narrator:** "Similarly, the freelancer creates their profile, associating it with existing skills."
    -
    - Run **`GET /FreelancerProfiles/me`**.
    - Run **`PUT /FreelancerProfiles/{id}`** using `{{freelancerProfileId}}` to update the bio.
    - Run **`POST /FreelancerProfiles/{id}/profile-picture`** with `{{freelancerProfileId}}` and a sample image file.
    - Run **`POST /FreelancerProfiles/{id}/resume`** with `{{freelancerProfileId}}` and a sample PDF file.
        - **Narrator:** "The API supports crucial file uploads for resumes and profile pictures, which are vital for a freelancer's identity."
    -
6.
7. **Public Profile Views (As Client):**
    - Run **`POST /Auth/login`** as `client@example.com`.
    - Run **`GET /ClientProfiles/{id}`** using `{{clientProfileId}}`.
    - Run **`GET /FreelancerProfiles/{id}`** using `{{freelancerProfileId}}`.
    - Run **`GET /FreelancerProfiles`**.

        - **Narrator:** "Now, let's switch back to the client. They can view any public profile, whether it's another client's or a freelancer's, and they can browse the entire list of available freelancers."
    -
8.

Act III: The Core Business - Projects & Proposals

Narrator: "Now we get to the heart of the platform: creating work and applying for it."

Operator Actions:

1. **`Projects` Folder (As Client):**
    - Run **`POST /Auth/login`** as `client@example.com`.
    - Run **`POST /Projects`**. Save the returned `id` as `projectId`.
        - **Narrator:** "Our client posts a new project, specifying the budget, deadline, and skills required."
    -
    - Run **`POST /Projects/{projectId}/attachments`** using `{{projectId}}` to upload a "Project Brief.pdf". Save the `id` as `attachmentId`.
        - **Narrator:** "They can attach important documents directly to the project."
    -
    - Run **`GET /Projects/{projectId}/attachments`**.
        - **Narrator:** "And view a list of all attachments for that project."
    -
    - Run **`GET /Projects/{id}`** using `{{projectId}}`.
        - **Narrator:** "They can view their own project details at any time."
    -
2.
3. **`Proposals` Folder (As Freelancer):**
    - Run **`POST /Auth/login`** as `freelancer@example.com`.
    - Run **`GET /Projects`** to see the new project.
    - Run **`GET /Projects/{projectId}/attachments/{attachmentId}`** to download the brief.
        - **Narrator:** "The freelancer logs in, sees the new project, and can download the project brief to review it."
    -
    - Run **`POST /Proposals`** using `{{projectId}}`. Save the returned `id` as `proposalId`.

        - **Narrator:** "After reviewing, they submit a proposal for the project."
    -
    - Run **`POST /Proposals/{proposalId}/attachments`** to add a "Portfolio-Excerpt.pdf".
        - **Narrator:** "They can even include their own attachments with their proposal."
    -
4.
5. **Project Management (As Client):**
    - Run **`POST /Auth/login`** as `client@example.com`.
    - Run **`GET /Proposals/for-project/{projectId}`**.
    - Run **`GET /Proposals/{proposalId}/attachments`**.
        - **Narrator:** "The client logs back in and can now see all proposals for their project, including any attachments."
    -
    - Run **`PUT /Proposals/{id}/status`** using `{{proposalId}}` to "Accept" it.
        - **Narrator:** "They accept the freelancer's proposal."
    -
    - Run **`PUT /Projects/{id}/assign-freelancer/{freelancerId}`**. (This could also be an automatic step after proposal acceptance, but we show the endpoint).
        - **Narrator:** "The client formally assigns the project to the freelancer."
    -
    - Run **`PUT /Projects/{id}/update`** using `{{projectId}}` to clarify a requirement. (This will fail now that the status isn't 'Open', demonstrating a business rule).
        - **Narrator:** "Note that certain actions, like updating the core project details, are now locked because the project is in progress. This demonstrates our API enforces the correct business logic."
    -
6.

Act IV: The Endgame - Completion, Ratings & Cleanup

Narrator: "Finally, we'll see the project through to completion and handle the cleanup and administrative tasks."

Operator Actions:

1. **Project Completion (As Client):**

- Run **`POST /Auth/login`** as `client@example.com`.
- Run **`PUT /Projects/{id}/mark-completed`**.
- Run **`PUT /Projects/{id}/cancel`** (This will fail, showing status logic).
  - **Narrator:** "Once work is done, the client marks the project as completed. The API prevents invalid state changes, like trying to cancel a completed project."
-

2.
3. **`Ratings` Folder (Both Roles):**
   - Run **`POST /Ratings`** where the client rates the freelancer. Save the `id` as `ratingId`.
   - Run **`POST /Auth/login`** as `freelancer@example.com`.
   - Run **`POST /Ratings`** where the freelancer rates the client.
     - **Narrator:** "A key feature is our two-way rating system, which is enabled only after a project is completed."
   -
   - Run **`GET /Ratings/{id}`** with `{{ratingId}}`.
   - Run **`GET /Ratings/received-by/{userId}`** with `{{freelancerId}}`.
   - Run **`GET /Ratings/given-by/{userId}`** with `{{clientId}}`.
   - Run **`GET /Ratings/average-for/{userId}`** with `{{freelancerId}}`.
     - **Narrator:** "We can then retrieve individual ratings, see all ratings given or received by a user, and calculate their average score, which is crucial for reputation."
   -
4.
5. **Cleanup & Deletion (As Owners & Admin):**
   - **As Freelancer:** Run **`POST /Auth/login`** as `freelancer@example.com`.
     - Run **`DELETE /FreelancerProfiles/{id}/resume`**.
     - Run **`DELETE /FreelancerProfiles/{id}/profile-picture`**.
     - Run **`DELETE /Proposals/{id}`** on another 'Pending' proposal to show it works.
   -
   - **As Client:** Run **`POST /Auth/login`** as `client@example.com`.
     - Run **`DELETE /Projects/{projectId}/attachments/{attachmentId}`**.
   -
   - **As Admin:** Run **`POST /Auth/login`** as `admin@example.com`.
     - Run **`DELETE /Projects/{id}`** with `{{projectId}}`.
     - Run **`DELETE /Ratings/{id}`** with `{{ratingId}}`.

        - Run **`DELETE /Skills/{id}`** with `{{skillId}}`.
        - Run **`DELETE /Users/{id}`** with `{{freelancerId}}`.
        - **Narrator:** "Finally, the API provides comprehensive delete operations, respecting user ownership and providing administrators with the power to manage and clean up any entity in the system, from project attachments to entire user accounts."
    -
6.

Narrator (Conclusion): "This concludes our exhaustive tour. As you've seen, every piece of functionality is exposed through a secure, logical, and well-defined API endpoint, providing a powerful foundation for any application built on this platform. Thank you."

The Live Showcase: A Step-by-Step Guide

(The demo begins. The Operator runs the Postman requests while the Narrator explains.)

Act I: The Foundation - Users, Auth, and Admin Oversight

Narrator: "Welcome. We'll begin by establishing the foundational user accounts for our platform: a Client, a Freelancer, and we'll then log in as an Administrator to see how they can manage the system."

Step	Operator Action (in Postman)	Narrator's Script
1	**`Auth` > `Register New Client`**: Run.	"First, a new client registers. The API validates the input and creates the account."
2	**`Auth` > `Register New Freelancer`**: Run.	"Next, a freelancer joins the platform through the same registration process."
3	**`Auth` > `Login as Admin`**: Run.	"Now, we'll log in as an Administrator. Watch the collection variables: our script automatically saves the Admin's JWT token for all future requests."
4	**`Users` > `Get All Users (Admin)`**: Run.	"As an Admin, we have a complete overview and can retrieve a paginated list of all users."
5	**`Users` > `Get User by ID (Client)`**: Run.	"The Admin can drill down to view the specific details of any user, like our newly created client. We'll save this client's ID for later."
6	**`Auth` > `Login as Client`**: Run.	"Let's switch roles. We're now logged in as the client. The JWT token has been automatically updated."

7	**`Users > Update User (Self)`**: Run.	"A user has full control over their own profile details, such as updating their name."
8	**`Users > Change Password (Self)`**: Run.	"And they can securely change their own password without involving an administrator."

Act II: Building Professional Identities

Narrator: "With accounts created, our users need to build their professional profiles. This involves creating profiles and managing the skills that define their work."

Step	Operator Action (in Postman)	Narrator's Script
9	**`Auth > Login as Admin`**: Run.	"First, the Admin curates the skills available on the platform."
10	**`Skills > Create Skill (Admin)`**: Run.	"Let's add a new skill called 'Postman Demo Skill'. The API returns the newly created skill, and we'll save its ID."
11	**`Skills > Get All Skills`**: Run.	"We can then retrieve the entire, filterable list of skills in the system."
12	**`Skills > Get Skill by ID`**: Run.	"And, of course, fetch any specific skill by its unique ID."
13	**`Auth > Login as Client`**: Run.	"Switching back to our client, they can now create their company profile."
14	**`Client Profiles > Create Client Profile`**: Run.	"The profile is created and linked to their user account. We'll save the new profile ID."
15	**`Client Profiles > Get My Client Profile`**: Run.	"A dedicated endpoint allows users to easily fetch their own profile."

16	**Client Profiles > Update Client Profile**: Run.	"And they can update their details at any time."
17	**Auth > Login as Freelancer**: Run.	"Now for the freelancer. After logging in, they'll create their profile."
18	**Freelancer Profiles > Create Freelancer Profile**: Run.	"The freelancer populates their profile with a headline, bio, and associates it with the skills we created earlier."
19	**Freelancer Profiles > Get My Freelancer Profile**: Run.	"Just like the client, they have a dedicated endpoint to view their own profile."
20	**Freelancer Profiles > Update Freelancer Profile**: Run.	"And they can update it to keep it fresh."
21	**Freelancer Profiles > Upload Profile Picture**: Run.	"Our API handles multipart file uploads, allowing the freelancer to upload a profile picture."
22	**Freelancer Profiles > Upload Resume**: Run.	"And a resume, both of which are critical for their identity on the platform."
23	**Auth > Login as Client**: Run.	"Let's log back in as the client to see what they can view."
24	**Client Profiles > Get Client Profile by ID**: Run.	"They can view other public profiles, including other clients."
25	**Freelancer Profiles > Get Freelancer Profile by ID**: Run.	"And more importantly, they can view our freelancer's newly created profile."
26	**Freelancer Profiles > Get All Freelancer Profiles**: Run.	"They can also browse and filter through the entire marketplace of available freelancers."

Act III: The Core Business Logic - Projects, Proposals, and Files

Narrator: "Now we get to the heart of the platform: a client creates a project, and a freelancer applies for it."

Step	Operator Action (in Postman)	Narrator's Script
27	**`Auth`** > **`Login as Client`**: Run.	"Our client is ready to post a job."
28	**`Projects`** > **`Create Project`**: Run.	"They create a new project, defining the scope, budget, and required skills. We save the new Project ID."
29	**`Projects`** > **`Upload Project Attachment`**: Run.	"To provide more detail, they upload a project brief as an attachment. We save the Attachment ID."
30	**`Auth`** > **`Login as Freelancer`**: Run.	"The freelancer logs in to find work."
31	**`Projects`** > **`Get All Projects`**: Run.	"They browse the project listings and find the new project from our client."
32	**`Projects`** > **`Get Project Attachments`**: Run.	"They can view the list of attachments for the project..."
33	**`Projects`** > **`Download Project Attachment`**: Run.	"...and download the project brief to review the requirements."
34	**`Proposals`** > **`Create Proposal`**: Run.	"Impressed, they submit a proposal with a cover letter and budget. We save the Proposal ID."

35	**`Auth` > `Login as Client`**: Run.	"The client logs back in to see if they've received any proposals."
36	**`Proposals` > `Get Proposals For Project`**: Run.	"They can view all proposals submitted for their specific project."
37	**`Proposals` > `Get Proposal by ID`**: Run.	"And they can drill down to see the details of our freelancer's proposal."
38	**`Proposals` > `Update Proposal Status`**: Run to "Accept" it.	"The client accepts the proposal. This triggers a critical state change in our system."
39	**`Projects` > `Get Project by ID`**: Run.	"If we now re-fetch the project, we'll see its status has automatically been updated to 'Assigned' and the freelancer's ID is now linked."
40	**`Projects` > `Mark Project Completed`**: Run.	"After the work is notionally completed, the client marks the project as finished."

Act IV: Resolution & Reputation - Ratings

Narrator: "With the project complete, the final step in the workflow is the two-way rating system, which builds trust and reputation on the platform."

Step	Operator Action (in Postman)	Narrator's Script
41	**`Auth` > `Login as Client`**: Run.	"The client will now rate the freelancer's performance."
42	**`Ratings` > `Create Rating (Client rates Freelancer)`**: Run.	"They submit a 5-star rating with a positive comment. We save the Rating ID."

43	**`Auth` > `Login as Freelancer`**: Run.	"Now the freelancer logs in to provide their feedback on the client."
44	**`Proposals` > `Get Proposals By Freelancer`**: Run.	"A freelancer can always review a list of all proposals they have submitted."
45	**`Ratings` > `Get Rating by ID`**: Run.	"Anyone involved can view the details of a specific rating."
46	**`Ratings` > `Get Ratings Received By User`**: Run for the Freelancer.	"We can easily query for all ratings a specific user has received."
47	**`Ratings` > `Get Ratings Given By User`**: Run for the Client.	"Or all the ratings they have given."
48	**`Ratings` > `Get Average Rating For User`**: Run for the Freelancer.	"And crucially, the API can calculate the average rating for any user, which is their public reputation score."

Act V: Security, Housekeeping, and Cleanup

Narrator: "Finally, we'll demonstrate the robustness of our security and the comprehensive cleanup capabilities available to users and administrators."

Step	Operator Action (in Postman)	Narrator's Script
49	**`Auth` > `Login as Freelancer`**: Run.	"Let's prove our role-based security works. We are logged in as a freelancer."
50	**`Projects` > `Create Project`**: Run.	"When the freelancer tries to create a project—a client-only action—the API correctly responds with a `403 Forbidden` error."

51	**Files > Get File Metadata**: Run.	"Our API also provides generic file management endpoints. Here we can fetch the metadata for the attachment we uploaded earlier."
52	**Files > Download File**: Run.	"And download it again, independent of the project context."
53	**Auth > Login as Admin**: Run.	"Now for the final cleanup, performed by the Admin."
54	**Files > Delete File (Admin)**: Run.	"The Admin can delete any file in the system."
55	**Ratings > Delete Rating (Admin)**: Run.	"They can moderate and remove ratings."
56	**Proposals > Delete Proposal**: Run.	"They can remove proposals."
57	**Projects > Delete Project (Admin)**: Run.	"And entire projects."
58	**Client Profiles > Delete Client Profile (Admin)**: Run.	"They have the power to delete user profiles."
59	**Freelancer Profiles > Delete Freelancer Profile (Admin)**: Run.	"Including freelancer profiles."
60	**Skills > Delete Skill (Admin)**: Run.	"They can curate the skills list by removing entries."
61	**Users > Delete User (Admin)**: Run.	"And finally, they can delete user accounts, completing the administrative lifecycle."

Narrator (Conclusion): "This concludes our comprehensive tour. We have successfully demonstrated all 61 endpoints, showcasing a complete, secure, and logical workflow from user creation to project completion and system administration. Thank you."