Real-Time Collaborative Workspace Backend

Backend Developer Assessment

Name: Aaditya Aaryan

Email: aadityaaryan639@gmail.com

Phone: +91 8340118693

Date: December 27, 2025

Table of Contents

1. Project Overview

This project is a Real-Time Collaborative Workspace Backend - a production-ready API service that enables teams to collaborate on projects in real-time. It features secure authentication, project and workspace management, role-based access control, and asynchronous code execution jobs.

Tech Stack:

- **Framework**: FastAPI (Python 3.11)
- **Databases**: PostgreSQL (relational data), MongoDB (job results & logs), Redis (caching & pub/sub)
- **Authentication**: JWT with Argon2 password hashing
- **Async Workers**: Celery for background job processing
- **Real-Time**: WebSocket with Redis Pub/Sub

Key Features:

- User registration, login, and profile management
- Project CRUD with collaborator invitations
- Workspace management within projects
- Role-based access control (Owner, Collaborator, Viewer)
- Async code execution with status tracking
- Real-time collaboration via WebSockets
- Rate limiting and API caching
- Feature flags for runtime configuration

2. Architecture Overview

The system follows a microservices-inspired architecture with clear separation of concerns:

System Components:

1. **FastAPI Application**: Handles HTTP/WebSocket requests, input validation, and routing

2. **PostgreSQL Database**: Stores users, projects, workspaces, and collaborator relationships

3. **MongoDB Database**: Stores high-velocity data like job results and activity logs

4. **Redis**: Serves as cache, rate limiter, pub/sub broker, and Celery message queue

5. **Celery Workers**: Process async jobs like code execution in the background

Request Flow:

1. Client sends request → Load Balancer → FastAPI instance

2. FastAPI validates input, checks auth (JWT), and rate limits (Redis)

3. Business logic executed, data fetched/stored in PostgreSQL/MongoDB

4. For async jobs: Task queued to Redis → Celery worker processes → Result stored in MongoDB

5. For real-time: WebSocket connections use Redis Pub/Sub for cross-instance messaging

Database Schema (PostgreSQL):

- `cw_users`: User accounts with hashed passwords
- `cw_projects`: Projects owned by users
- `cw_workspaces`: Workspaces within projects
- `cw_collaborators`: Many-to-many relationship with roles

Key Libraries:

- SQLAlchemy 2.0 with async support for PostgreSQL
- Motor for async MongoDB operations
- redis-py for async Redis operations
- passlib + argon2-cffi for password hashing
- PyJWT for token management

3. Setup & Run Instructions

Prerequisites:

- Python 3.11+
- PostgreSQL, MongoDB, Redis (or use Docker)

Local Setup:

1. Clone the repository:

```
git clone https://github.com/Aadik1ng/Collaborative-Workflow.git
cd Collaborative-Workflow
```

2. Create virtual environment:

```
python -m venv venv
source venv/bin/activate # Windows: venv\Scripts\activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

4. Configure environment variables:

```
cp .env.example .env
# Edit .env with your database credentials
```

5. Start the API server:

```
uvicorn app.main:app --reload
```

6. Start Celery worker (separate terminal):

```
celery -A app.workers.celery_app worker --loglevel=info
```

Docker Setup:

```
cd docker
docker-compose up -d
```

Environment Variables Required:

- POSTGRES_URL: PostgreSQL connection string (with +asyncpg)
- MONGODB_URL: MongoDB connection string
- MONGODB_DATABASE: Database name
- REDIS_URL: Redis connection string
- SECRET_KEY: JWT signing secret
- ALGORITHM: JWT algorithm (HS256)
- ACCESS_TOKEN_EXPIRE_MINUTES: Token lifetime
- REFRESH_TOKEN_EXPIRE_DAYS: Refresh token lifetime

4. API Documentation

Authentication Endpoints

POST /api/v1/auth/register

Description: Register new user

Request: `{"email": "user@example.com", "username": "user", "password": "SecurePass123!", "full_name": "John Doe"}`

Response: `{"id": "uuid", "email": "...", "username": "...", "full_name": "..."}`

Status: 201 Created

POST /api/v1/auth/login

Description: Login and get tokens

Request: `{"email": "user@example.com", "password": "SecurePass123!"}`

Response: `{"access_token": "...", "refresh_token": "...", "token_type": "bearer"}`

Status: 200 OK

POST /api/v1/auth/refresh

Description: Refresh access token

Request: `{"refresh_token": "..."}`

Response: `{"access_token": "...", "token_type": "bearer"}`

Status: 200 OK

POST /api/v1/auth/logout

Description: Logout user

Request: `Header: Authorization: Bearer <token>`

Response: `{"message": "Logged out successfully"}`

Status: 200 OK

GET /api/v1/auth/me

Description: Get current user

Request: `Header: Authorization: Bearer <token>`

Response: `{"id": "...", "email": "...", "username": "...", "full_name": "..."}`

Status: 200 OK

PUT /api/v1/auth/me

Description: Update profile

Request: `{"full_name": "New Name"}`

Response: `{"id": "...", "email": "...", "full_name": "New Name"}`

Status: 200 OK

Project Endpoints

POST /api/v1/projects

Description: Create project

Request: `{"name": "My Project", "description": "...", "is_public": false}`

Response: `{"id": "uuid", "name": "...", "owner_id": "...", "created_at": "..."}`

Status: 201 Created

GET /api/v1/projects

Description: List projects

Request: `Query: ?skip=0&limit;=10`

Response: `[{"id": "...", "name": "...", ...}]`

Status: 200 OK

GET /api/v1/projects/{id}

Description: Get project

Request: `Path: project ID`

Response: `{"id": "...", "name": "...", "owner": {...}, "workspaces": [...]}`

Status: 200 OK

PUT /api/v1/projects/{id}

Description: Update project

Request: `{"name": "Updated Name"}`

Response: `{"id": "...", "name": "Updated Name", ...}`

Status: 200 OK

DELETE /api/v1/projects/{id}

Description: Delete project

Request: `Path: project ID`

Response: `{"message": "Project deleted"}`

Status: 200 OK

Workspace Endpoints

POST /api/v1/projects/{id}/workspaces

Description: Create workspace

Request: `{"name": "Workspace 1", "description": "..."}`

Response: `{"id": "uuid", "name": "...", "project_id": "..."}`

Status: 201 Created

GET /api/v1/projects/{id}/workspaces

Description: List workspaces

Request: `Path: project ID`

Response: `[{"id": "...", "name": "...", ...}]`

Status: 200 OK

Collaborator Endpoints

POST /api/v1/projects/{id}/collaborators

Description: Invite collaborator

Request: `{"email": "collab@example.com", "role": "collaborator"}`

Response: `{"id": "...", "user_id": "...", "role": "collaborator"}`

Status: 201 Created

GET /api/v1/projects/{id}/collaborators

Description: List collaborators

Request: `Path: project ID`

Response: `[{"user_id": "...", "email": "...", "role": "..."}]`

Status: 200 OK

Job Endpoints

POST /api/v1/jobs

Description: Submit code execution job

Request: `{"language": "python", "code": "print('Hello')", "timeout": 30}`

Response: `{"id": "uuid", "status": "pending", "created_at": "..."}`

Status: 202 Accepted

GET /api/v1/jobs/{id}

Description: Get job status

Request: `Path: job ID`

Response: `{"id": "...", "status": "completed", "output": "Hello", "execution_time": 0.5}`

Status: 200 OK

GET /api/v1/jobs

Description: List user jobs

Request: `Query: ?skip=0&limit;=10`

Response: `[{"id": "...", "status": "...", ...}]`

Status: 200 OK

POST /api/v1/jobs/{id}/cancel

Description: Cancel job

Request: `Path: job ID`

Response: `{"message": "Job cancelled"}`

Status: 200 OK

5. Design Decisions & Trade-offs

1. Dual Database Strategy (PostgreSQL + MongoDB)

- Rationale: Relational data (users, projects, roles) benefits from ACID transactions. Non-relational data (job results, logs) needs flexible schemas and high write throughput.

- Trade-off: Increased operational complexity.

2. Argon2 for Password Hashing

- Rationale: Winner of Password Hashing Competition, resistant to GPU attacks, no 72-byte limit like bcrypt.

- Trade-off: Slightly higher CPU usage per hash.

3. Celery for Async Jobs

- Rationale: Decouples long-running tasks from request cycle, improves API responsiveness.

- Trade-off: Adds Redis as required dependency.

4. JWT with Refresh Tokens

- Rationale: Stateless access tokens enable horizontal scaling. Refresh tokens allow session invalidation.

- Trade-off: Requires careful token handling on client.

5. Table Name Prefixing (cw_)

- Rationale: Allows coexistence with other apps in shared database.

- Trade-off: Longer table names.

6. Direct argon2-cffi Usage

- Rationale: Bypasses passlib's backend detection issues in serverless environments.

- Trade-off: Less abstraction.

6. Scalability Considerations

Horizontal Scaling:

• FastAPI: Deploy multiple instances behind load balancer (stateless design)

• Celery Workers: Add more workers for increased job throughput

• PostgreSQL: Use read replicas, connection pooling (PgBouncer)

• MongoDB: Sharding for write scaling, replica sets for reads

• Redis: Redis Cluster for HA and scaling

Performance Optimizations:

• Sliding-window rate limiter protects against abuse

• Redis caching for frequently accessed data

• SQLAlchemy async connection pooling

• Idempotent job processing with unique IDs

Database Indexing:

• Indexed: user email, username, project owner_id, workspace project_id

• TTL indexes on activity logs (7-day expiry)

Security Measures:

• Argon2 password hashing

• JWT with short-lived access tokens

• Input validation with Pydantic

• CORS configuration

• Rate limiting per IP/user

Future Enhancements:

• Code execution sandboxing (Docker/gVisor)

• OpenTelemetry for distributed tracing

• Kubernetes deployment with Helm charts

7. Testing Instructions

Running Tests:

```
# Run all tests
pytest

# Run integration tests
pytest tests/integration

# Run unit tests
pytest tests/unit

# Run with coverage
pytest --cov=app --cov-report=term-missing

# Run with verbose output
pytest -v
```

Test Coverage:

- 58 tests covering authentication, projects, workspaces, collaborators, jobs
- ~59% code coverage focusing on critical paths
- Uses SQLite in-memory for fast integration tests

Test Categories:

- Unit Tests: Password hashing, JWT tokens, permissions
- Integration Tests: Full API request/response cycles with mocked databases

8. Deployment Instructions

Vercel Deployment:

1. Connect your GitHub repository to Vercel

2. Select "Other" as framework preset

3. Set environment variables in Vercel Dashboard:

- POSTGRES_URL (with +asyncpg prefix)

- MONGODB_URL

- MONGODB_DATABASE

- REDIS_URL

- SECRET_KEY

- Other JWT/app settings

4. Override Install Command: pip install -r requirements.txt

5. Deploy - Vercel auto-deploys on push to main

Important Notes:

• Vercel serverless has timeouts; use persistent hosting for WebSockets

• Celery workers must be deployed separately (Railway, Heroku, VPS)

• Use external databases (Railway, Atlas, Upstash)

Docker Deployment:

```
cd docker
docker-compose up -d --build
```

Railway Deployment:

1. Create PostgreSQL, MongoDB, Redis services

2. Deploy API from GitHub

3. Deploy Celery worker as separate service

4. Configure environment variables

9. Links Summary

GitHub Repository: https://github.com/Aadik1ng/Collaborative-Workflow

Live Vercel Deployment: https://collaborative-workflow.vercel.app

Walkthrough Video: https://your-video-link.com