Overview of Phase 2: Backend Core APIs (Weeks 3-5)

Phase 2 is the core development stage of the project, focusing on building the foundational backend APIs that power the application’s key functionalities. This phase spans Weeks 3 through 5, assuming a standard 8-week project timeline as outlined in the document. The primary goal is to create a robust, secure, and scalable backend system using APIs to handle user interactions, data management, and collaboration features.

At a high level, this phase shifts from the planning and architecture setup in Phase 1 to actual implementation. You’ll be developing RESTful APIs (likely using a framework like FastAPI or Flask in Python, based on the tools mentioned like Pytest and Swagger/OpenAPI). The emphasis is on security (via authentication and access controls), data handling (CRUD operations for tenders and related entities), and performance optimizations (like caching).

By the end of Phase 2, you should have a working backend that can be tested independently and integrated with frontend or other components in later phases. This phase sets the stage for Phase 3’s AI integrations, so ensure the APIs are modular and extensible.

Now, let’s break it down extensively: the goals, core features, testing, additions, milestones, and a step-by-step guide on how to approach implementation.

Goals

The main objective is to implement essential APIs for:

• User Authentication: Secure login and session management.

• Tender Upload: Allowing users to submit and manage tender-related data.

• Profile Setup: User profiles with specific scoring metrics.

• Team Collaboration: Tools for teams to work together on projects.

These APIs form the “core” of the backend, meaning they’re the minimum viable features needed for the system to function. Focus on making them reliable, as they’ll handle sensitive data like documents and user profiles.

Core Features

Here’s a detailed explanation of each feature, including what it entails and why it’s important:

1 JWT-based Authentication & Role-Based Access Control (RBAC):

◦ Explanation: JWT (JSON Web Tokens) is a standard for securely transmitting information between parties as a JSON object. It’s stateless, meaning the server doesn’t need to store session data—everything is in the token. RBAC assigns permissions based on roles:

▪ Admin: Full access, e.g., managing users or approving tenders.

▪ SME (Subject Matter Expert): Likely focused on reviewing or scoring tenders.

▪ Collaborator: Limited to viewing/editing shared projects.

◦ Why it’s important: Security is critical for a tender management system to prevent unauthorized access to sensitive documents or profiles.

◦ Implementation details: Use libraries like PyJWT for token generation/validation. Integrate with a user database to store roles. Endpoints might include /login (to generate JWT) and middleware to validate tokens on protected routes.

2 Tender CRUD Operations (Metadata, Documents):

◦ Explanation: CRUD stands for Create, Read, Update, Delete. For tenders, this means:

▪ Create: Upload new tender metadata (e.g., title, deadline, description) and associated documents (e.g., PDFs via file uploads).

▪ Read: Retrieve tender details.

▪ Update: Edit metadata or replace documents.

▪ Delete: Remove tenders (with soft-delete options for safety).

◦ Why it’s important: Tenders are the central entity in this system (likely a procurement or bidding platform), so efficient management is key.

◦ Implementation details: Use a database schema from Phase 1. Store metadata in structured fields and documents in blob storage (e.g., AWS S3 or local files). APIs like /tenders (GET for list, POST for create), /tenders/{id} (GET/PUT/DELETE).

3 Team Workspace Setup (Project Links, Shared Notes):

◦ Explanation: This creates collaborative spaces where teams can link projects to tenders, share notes, and possibly discuss in real-time or asynchronously.

▪ Project links: URLs or references to related tenders/projects.

▪ Shared notes: Text fields or documents that multiple users can edit.

◦ Why it’s important: Enables teamwork, which is essential for complex tender processes involving multiple stakeholders.

◦ Implementation details: Model this as a “workspace” entity in the database, linked to users via many-to-many relationships. APIs like /workspaces for creation/listing, /workspaces/{id}/notes for sharing.

4 Profile Setup & Scoring Inputs (e.g., CIDB, BEE Level):

◦ Explanation: Users set up profiles with data like CIDB (Construction Industry Development Board) grades or BEE (Broad-Based Black Economic Empowerment) levels, which are South African compliance metrics for tenders. These could be used for automated scoring in Phase 3.

◦ Why it’s important: Profiles personalize the experience and feed into analytics/readiness scoring.

◦ Implementation details: A /profiles endpoint for CRUD on user profiles. Store scoring inputs as fields in the user table (e.g., integers for levels).

5 Public API Endpoint for Listing Tenders:

◦ Explanation: An open endpoint (no auth required) to list available tenders, perhaps filtered by criteria like status or category.

◦ Why it’s important: Allows external users or integrations to browse tenders without logging in, increasing accessibility.

◦ Implementation details: A simple GET /public/tenders endpoint, possibly paginated for performance.

Testing

Testing ensures the APIs are bug-free and perform as expected:

• Swagger/OpenAPI Documentation: Use tools like FastAPI’s built-in Swagger UI or OpenAPI specs to auto-generate interactive docs. This allows easy testing of endpoints via a web interface.

• Pytest Unit Tests: Write tests for each endpoint’s logic (e.g., test auth failures, successful CRUD). Pytest is a Python testing framework—focus on edge cases like invalid inputs or role-based denials.

Aim for 80%+ code coverage. Include integration tests to simulate real API calls.

Additions

These are enhancements to make the backend more efficient:

• Redis Caching for Public Tenders API: Redis is an in-memory data store for caching. Cache the results of the public tenders list to reduce database queries and speed up responses (e.g., cache for 5-10 minutes).

◦ Why: Public endpoints might get high traffic; caching prevents overload.

• PostgreSQL JSONB for Semi-Structured Fields: Use JSONB columns in PostgreSQL for flexible data like custom tender metadata (e.g., arbitrary key-value pairs).

◦ Why: Allows storing unstructured data without rigid schema changes.

Milestones

• Backend Milestone Review: At the end of Week 4 or mid-phase, review the implemented features with stakeholders for feedback.

• API Integration Tests: End-of-phase tests to ensure APIs work together (e.g., auth + tender CRUD).

These act as checkpoints to catch issues early.

How to Go About Implementing Phase 2: Step-by-Step Guide

Approach this phase methodically to stay on schedule (3 weeks total). Assume a team setup: backend developers, possibly with a lead for reviews. Use Agile practices like daily standups and sprints.

1 Preparation (Start of Week 3):

◦ Review Phase 1 outputs: Architecture, schema, and NFRs (e.g., performance targets like <100ms response time).

◦ Set up the development environment: Install PostgreSQL, Redis, and your API framework (e.g., FastAPI). Create a Git repo for version control.

◦ Define API specs: Use OpenAPI to outline endpoints, request/response schemas, and error handling.

2 Implementation Breakdown (Weeks 3-4):

◦ Week 3: Auth and Profiles:

▪ Build JWT auth first—it’s a dependency for other features.

▪ Implement profile setup APIs next, as they’re user-centric.

▪ Test incrementally with Pytest.

◦ Week 4: Tenders and Collaboration:

▪ Develop tender CRUD, integrating document uploads (use libraries like boto3 for storage).

▪ Add team workspace features.

▪ Implement the public tenders endpoint with Redis caching (use redis-py library).

▪ Integrate JSONB fields in your database models (use SQLAlchemy ORM for Python).

3 Testing and Documentation (Mid-Week 4 to Week 5):

◦ Write Swagger docs as you build endpoints.

◦ Run unit and integration tests. Use tools like Postman for manual API testing.

◦ Handle security: Implement rate limiting, input validation (e.g., via Pydantic), and error logging.

4 Reviews and Iterations (End of Week 5):

◦ Conduct the backend review: Demo APIs, share code for peer review.

◦ Fix bugs based on feedback.

◦ Ensure scalability: Test with mock data (e.g., 1000 tenders) to check performance.

5 Best Practices and Tips:

◦ Tech Stack: Python-based (inferred from Pytest). Use Docker for containerization to ease deployment.

◦ Security: Always hash passwords (e.g., bcrypt), use HTTPS, and validate JWTs properly.

◦ Error Handling: Standardize responses (e.g., JSON with status codes).

◦ Collaboration: Use tools like GitHub for code reviews and Jira/Slack for tracking.

◦ Time Management: Allocate 40% to coding, 30% to testing, 20% to docs/reviews, 10% buffer for surprises.

◦ Potential Challenges: File uploads can be tricky—handle large files with streaming. Ensure RBAC doesn’t leak data.

◦ Resources: Refer to official docs for JWT, PostgreSQL, Redis. If stuck, prototype small parts first.

Tasks:

Koketso- Public API endpoint for listing tenders

Snethemba- JWT-based Authentication & Role-Based Access Control (RBAC):

Ashwil- Tender CRUD Operations (Metadata, Documents):

Onthatile- Team Workspace Setup (Project Links, Shared Notes):

Khethiwe- Profile Setup & Scoring Inputs (e.g., CIDB, BEE Level):

Report:  
Everyone submitted what they had to do on github repository but it was not fully functional.  
Meetings:

25 August 2025: Everyone present, project phase 2 scope and deliverables clearly indicated

17 September 2025- meeting at the library, only Koketso, Khethiwe and Ashwil showed up. Sinethemba issues an apology.